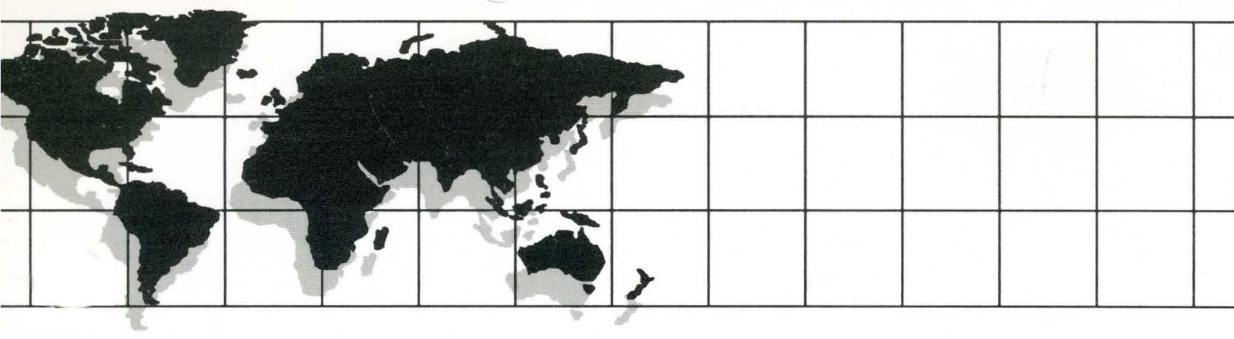


CTOS

Procedural Interface

Reference Manual

Volume 2
Operations H through R

UNISYS

UNISYS

CTOS®
Procedural Interface
Reference Manual
Volume 2
Operations H through R

This volume has been updated for CTOS III with pages from
Update 1, 4357 4714-110

CTOS III 1.0	August 1992
CTOS II 3.4	
CTOS I 3.4	
CTOS/XE 3.4	
Development Utilities 12.2	
Standard Software12.2	
Video Access Method 4.0	Printed in USA
Priced Item	4357 4656-100

Contents

List of Figures

List of Tables

HeapAlloc(cBytes, poMemoryRet): ercType

Description

HeapAlloc is used by an asynchronous system service program to allocate memory from the heap. Rather than storing all variables in the stack, it is more efficient for your program to use this operation to allocate storage for large blocks of data as the storage space is needed. The asynchronous procedure HeapFree can then be used to deallocate the memory when it is no longer needed.

HeapAlloc returns status code 4533 ("No heap memory available") if the heap does not have not enough contiguous memory to meet the requirements of the request.

Procedural Interface

HeapAlloc(cBytes, poMemoryRet): ercType

where

cBytes

is the count of bytes to allocate from the heap.

poMemoryRet

is the memory address to which the pointer to the allocated memory is returned.

Request Block

HeapAlloc is an object module procedure in Async.lib.

This page intentionally left blank

HeapFree(pMemory): ercType

Description

HeapFree is used by an asynchronous system service to return memory to the heap. The memory must have been previously allocated by a call to the HeapAlloc operation.

HeapFree returns status code 4534 ("Invalid heap block") if the pointer passed to it is not the address of a valid heap block.

Procedural Interface

HeapFree(pMemory): ercType

where

pMemory

is the pointer to the block that was previously allocated from the heap.

Request Block

HeapFree is an object module procedure in Async.lib.

This page intentionally left blank

HeapInit (cBytes, pHeap): ercType

Description

HeapInit allocates a fixed amount of memory for the heap used by an ansynchronous system service. The heap must be initialized by this operation before any of the other asynchronous system service operations are used.

Procedural Interface

HeapInit (cBytes, pHeap): ercType

where

cBytes

is is the count of bytes to be used for the heap.

pHeap

is the pointer to the first byte of heap space.

Request Block

HeapInit is an object module procedure in Async.lib.

This page intentionally left blank

InitAltMsgFile (pMwa, sMwa, pbMsgFile, cbMsgFile, pbPassword, cbPassword, pBuffer, sBuffer, pbCache, cbCache): ercType

Description

InitAltMsgFile opens a binary message file of the form {Node}[VolName]<DirName>FileName for subsequent retrieval of numbered messages by operations, such as PrintAltMsg, GetAltMsg, and GetAltMsgUnexpanded. The message file must have been created in the Executive using the **Create Message File** command. (See the *CTOS Executive Reference Manual* for details.)

InitAltMsgFile is similar to InitMsgFile except that, with alternate message files, any number of message files may be open at any time. Access to each message file is determined by the Message Work Area (MWA).

Procedural Interface

InitAltMsgFile (pMwa, sMwa, pbMsgFile, cbMsgFile, pbPassword, cbPassword, pBuffer, sBuffer, pbCache, cbCache): ercType

where

pMwa

is the memory address of the MWA.

sMwa

is the size (word) of the MWA. The size must be at least 60 bytes.

pbMsgFile
cbMsgFile

describe the character string containing the name of the binary message file.

pbPassword
cbPassword

describe the password authorizing access to the message file.

pBuffer
sBuffer

describe a word-aligned I/O buffer. *sBuffer* must be at least 1024 bytes and must be a multiple of 512.

pbCache
cbCache

describe a word-aligned cache for the lookup table portion of the message file. *cbCache* must be at least 1024 bytes and must be a multiple of 512.

Request Block

InitAltMsgFile is an object module procedure.

InitCharMap (pMap, sMap): ercType

Description

InitCharMap initializes the character map. The ResetVideo and InitVidFrame operations must be called first.

InitCharMap sets all character positions of the character map to blanks and resets all line and character attributes. It then places the border character at the character positions that define the border of the frames for which borders were requested. The border descriptor, border character, and border attributes of each frame are specified by the InitVidFrame operation and are stored in a frame descriptor of the Video Control Block.

Procedural Interface

InitCharMap (pMap, sMap): ercType

where

pMap

is two words, both of which are 0 for compatibility with earlier versions of the operating system (all workstations). Status code 501 ("Invalid argument to VDM") is returned if this parameter is omitted.

sMap

is the byte size of the character map as returned by the *psMapRet* parameter of the ResetVideo operation.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	76
12	reserved	6	
18	pMap	4	
22	sMap	2	

pbClcb
cbClcb

describe the Communications Line Configuration Block (CLCB). The format of the CLCB is shown below:

Offset	Field	Size (bytes)	Description
0	pDsBx	4	is an address placed in DS:BX upon entry to interrupt service routine.
4	pTxIsr	4	is the address of the transmit interrupt procedure.
8	pExtIsr	4	is the address of the external/status interrupt procedure.
12	pRxIsr	4	is the address of the receive interrupt procedure.
16	pSpRxIsr	4	is the address of the special receive interrupt procedure.
20	fRawTx	1	is a flag. TRUE means the transmit is a raw interrupt.
21	fRawExt	1	is a flag. TRUE means the external/status is a raw interrupt.
22	fRawRx	1	is a flag. TRUE means the receive is raw interrupt.
23	fRawSpRx	1	is a flag. TRUE means the special receive is a raw interrupt.
24	baudXmit	2	is the transmit baud rate (word), for example 19200.
26	baudRcv	2	is the receive baud rate (word), for example 19200.
28	fNRZI	1	is a flag. TRUE selects NRZI encoding.

Offset	Field	Size (bytes)	Description
29	fX21	1	is one of the following values: **Value** — **Description** 170 — selects X.21 port with 1984 protocol hardware 255 — selects X.21 drivers only 0 — does not select X.21 drivers (see the *CTOS Programming Guide)*
30	fTdi	1	is a flag. TRUE selects TDI levels.
32	fTdiXlat	1	is a flag. TRUE translates upstream TDI port to RS-232-C downstream.
32	fDMA	1	is a flag. TRUE selects DMA mode, if it is available.
33	fPtrXlat	1	is a flag. TRUE means the application is requesting the InitCommLine client to supply a printer translation.
34	fReserved	1	is a 1-byte flag reserved for future use. It must always be set to 0.
35	fRealMode	1	is a flag. TRUE means the application is executing in real mode. Otherwise the application is executing in protected mode.

Offset	Field	Size (bytes)	Description
36	fV35Mode	1	is a flag. TRUE means the application is requesting V35 drivers. The return information will indicate if V35 drivers are available. FALSE means that V35 drivers *must not* be used. An error is returned if operation in RS-232-C mode is not possible. If this parameter is not provided, the drivers being used will not be checked.

If a CLCB option is selected that is not supported by the selected hardware channel in the device specification provided, status code 7 ("Not implemented") is returned. Some channels do not support NRZI. For example, the XC-002 supports NRZI only on channels A and B. On some channels the transmit and receive baud rates must match. For example, the XC-002 supports separate baud rates for receive and transmit.

Baud rates may be zero (specifies external clocking). Some channels, such as those for a shared resource processor, allow external clocking of transmit while using internal clocking of receive, or vice versa; other channels (currently, all channels on workstations or XC-002) do not.

NOTE: On channels A and B of the SuperGen Series 2000 workstation, and on channel C of the SuperGen Series 5000 workstation, baud rates of 50, 1800, 3600, and 7200 are not supported.

The CLCB is designed to be expanded by addition of new options at the end, if necessary, to support future workstation modules or new workstations whose serial ports have special features executing in protected mode.

NOTE: InitCommLine complies with the CTOS/Open standard. For details on CTOS/Open, consult CTOS/Open Programming Practices and Standards.

pbRet
cbRet

describe the memory area into which the Communications Line Return Block (CLRB) is written. If either the 8274 or 8530 chip is present, the CLRB has the following format:

Offset	Field	Size (bytes)	Description
0	commLineHandle	2	is the communications line handle to be provided in subsequent communications operations such as ReadCommLineStatus or ResetCommLine.
2	ioCommCtl	2	is the port address of the serial communications controller (SCC) control/ status register
4	ioCommData	2	is the port address of the SCC data register
6	chipType	2	is the chip type. The values of *chipType* are
			Value **Description**
			0 Intel 8274 chip type
			1 Intel 8530 chip type
8	pitResolution	2	is a value that, multiplied with the Programmable Interval Timer resolution, gives 1 millisecond

Offset	Field	Size (bytes)	Description
10	dmaAvailable	2	is set to TRUE by the operating system if DMA is available
12	fProvideXlatArea	1	is a flag that is TRUE if a translation buffer has been provided
13	pXlatarea	4	if the memory address of the translation area
17	cbXlatarea	2	is the size of translation area
19	ioX21	2	is the hardware status port for 1984 X.21 hardware (or is 0 if invalid)
21	lineType	2	is one of the following values:

Value	Description
0	channel A on the SCC is used
1	channel B on the SCC is used

Offset	Field	Size (bytes)	Description
23	fV35Avail	1	is a flag. TRUE means V.35 drivers are used.

If the 2681 chip is present, the returned CLRB has the following format:

Offset	Field	Size (bytes)	Description
0	commLineHandle	2	is the communications line handle to be provided in subsequent communications operations such as ReadCommLineStatus or ResetCommLine.
2	ioStatusReg	2	is the port address of the 2681 control/status register.

Offset	Field	Size (bytes)	Description
4	ioDataReg	2	is the port address of the 2681 data register.
6	chipType	2	is the chip. type. If the 2681 chip is present, *chipType* is 3.
8	wNU	2	is reserved.
10	ioCommandReg	2	is the port address of the 2681 command register. This register selects the register to write to, resets receiver, transmitter, and error status, start and stop break sequence, and disable/enable receiver/transmitter.
12	ioModeReg	2	is the port address of the 2681 mode register. This register sets character length, parity, stop bits, channel mode, RTS control, and RX INT select.
14	ioIntMask	2	is the register that enables or disables interrupt sources.
16	ioModemStatIn	2	is the output port config register and input to read the status of CTS and DSR.
18	ioModemOutMask	2	is the port to write to to reset DTR and RTS.
20	ioModemStatOut	2	is the port to write to to set DTR and RTS.

If the 16450 or 16551 chip is present, the returned CLRB has the following format:

Offset	Field	Size (bytes)	Description
0	commLineHandle	2	is the communications line handle to be provided in subsequent communications operations such as ReadCommLineStatus or ResetCommLine.
2	ioStatusReg	2	is the port address of the 16450 or 16551 status register.
4	ioDataReg	2	is the port address of the 16450 or 16551 data register.
6	chipType	2	is the chip type. The values of *chipType* are

Value	Description
4	Western Digital 16450 chip type
5	Western Digital 16551 chip type

Offset	Field	Size (bytes)	Description
8	ioCommandReg	2	is the port address of the 16450 or 16551 command register. This sets character length, stop bits, parity, and set break control.
10	ioIntMask	2	is the register that enables or disables interrupt sources.
12	ioModemCnt	2	is the register that sets the DTR, RTS, and so forth.
14	ioModemStat	2	is the register that reports the state of the CTS, DSR, RI, and so forth.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	2
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	4102
12	unused	6	
18	pbSpec	4	
20	cbSpec	2	
24	pbClcb	4	
28	cbClcb	2	
30	pbRet	4	
34	cbRet	2	

InitErcFile (pBuffer, sBuffer, pbCache, cbCache): ercType

Description

InitErcFile opens the binary error message file [Sys]<Sys>ErcMsg.bin (distributed with Standard Software) for subsequent retrieval of status code text by operations, such as GetErc, GetErcUnexpanded, and GetFileErc. If the error message file is not present, the file is opened on [!Sys]<Sys>.

InitErcFile is similar to InitMsgFile except that, with InitErcFile, the messages are retrieved from the error message file [Sys]<Sys>ErcMsg.bin rather than from the application's normal message file. This allows access to status code text without disturbing the application's message file.

Procedural Interface

InitErcFile (pBuffer, sBuffer, pbCache, cbCache): ercType

where

pBuffer
sBuffer

describe a word-alignedbuffer for I/O. *sBuffer* must be at least 1024 bytes and must be a multiple of 512.

pbCache
cbCache

describe a word-aligned cache for the lookup table portion of the error message file. *cbCache* must be at least 1024 bytes and must be a multiple of 512.

Request Block

InitErcFile is an object module procedure.

This page intentionally left blank

InitLargeOverlays (pSwapBuffer, cParasSwapBuffer): ercType

Description

InitLargeOverlays initializes the Virtual Code facility. It is identical to InitOverlays except that the size of the overlay buffer is described as a count of 16-byte paragraphs instead of as a count of bytes. (See InitOverlays.)

Using this operation, you can have an overlay buffer that is up to 1M-byte long.

Procedural Interface

InitLargeOverlays (pSwapBuffer, cParasSwapBuffer): ercType

where

pSwapBuffer

is the memory address of the first byte of the overlay buffer. The buffer must be word-aligned.

cParasSwapBuffer

is the size of the overlay buffer in 16-byte paragraphs. If your program runs in real mode, the buffer must be large enough to contain the largest nonresident code segment. If your program runs in protected mode, the buffer must be large enough to contain the largest nonresident code segment as well as the second largest nonresident code segment.

Request Block

InitLargeOverlays is an object module procedure.

This page intentionally left blank

InitLocalPageMap (pLocalPageMap):ercType

Caution: *This operation is for* **restricted use only** *and is subject to change in future operating system releases. It is supported in protected mode only on operating systems running on 80386 or later processors.*

Description

InitLocalPageMap initializes the segment addressed by *pLocalPageMap* for use as a local page map. The segment must be at least 8K bytes in size and must be aligned on a 4K-byte boundary in physical memory.

The AllocMemoryFramesSL operation allocates memory with the required alignment.

Once initialized, a local page map is defined by the DefineLocalPageMap operation.

InitLocalPageMap supports system software, such as the PC Emulator, which creates and maintains page maps on the 80386 microprocessor.

For details on page maps, see the *80386 Programmer's Reference Manual.*

Procedural Interface

InitLocalPageMap (pLocalPageMap):ercType

where

pLocalPageMap

is the memory address of the segment to be used as the local page map.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	2	4
2	nReqPbCb	1	0
3	nRespPbCp	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	326
12	pLocalPageMap	4	

InitMsgFile (pbMsgFile, cbMsgFile, pbPassword, cbPassword, pbBuffer, cbBuffer, pbCache, cbCache): ercType

Description

InitMsgFile opens a binary message file for subsequent retrieval of numbered messages. The message file must have been created in the Executive using the **Create Message File** command. (See the *CTOS Executive Reference Manual* for details.)

Procedural Interface

InitMsgFile (pbMsgFile, cbMsgFile, pbPassword, cbPassword, pbBuffer, cbBuffer, pbCache, cbCache): ercType

where

pbMsgFile
cbMsgFile

describe a character string of the following form: {Node}[Volume]<Dir>FileName. It contains the name of the binary message file to be opened.

pbPassword
cbPassword

describe the password authorizing access to the message file.

pbBuffer
cbBuffer

describe a word-aligned I/O buffer. *cbBuffer* must be at least 1024 bytes and must be a multiple of 512.

pbCache
cbCache

describe a word-aligned cache for the lookup table portion of the message file. *cbCache* must be at least 1024 bytes and must be a multiple of 512.

Request Block

InitMsgFile is an object module procedure.

InitOverlays (pSwapBuffer, sSwapBuffer): ercType

Description

InitOverlays initializes the Virtual Code facility. This operation must be included in the resident code of a program. It is called once at the beginning of a program and must be called before any operation in a nonresident (virtual) code segment is called.

InitOverlays is identical to InitLargeOverlays except that the size of the buffer is described as a count of bytes instead of as a count of 16–byte paragraphs.

Procedural Interface

InitOverlays (pSwapBuffer, sSwapBuffer): ercType

where

pSwapBuffer

is the memory address of the first byte of the overlay buffer. The buffer must be word-aligned.

sSwapBuffer

is the size of the overlay buffer. If your program runs in real mode, the buffer must be large enough to contain the largest nonresident code segment. If your program runs in protected mode, the buffer must be large enough to contain the largest nonresident code segment as well as the second largest nonresident code segment.

Request Block

InitOverlays is an object module procedure.

This page intentionally left blank

InitSysCmds (pbCmdFile, cbCmdFile): ercType

Description

InitSysCmds opens an Executive command file of the form {Node}[Volume]<Directory>FileName for the subsequent retrieval of command information. The command file must have been created using either the **Command File Editor** or the **New Command** command. (See the *CTOS Executive Reference Manual* for details.)

Once an application opens a command file, it can access the file by calling GetSysCmdInfo.

Before an application opens a second command file, it must use CloseSysCmds to close the command file that is currently open.

Procedural Interface

InitSysCmds (pbCmdFile, cbCmdFile): ercType

where

pbCmdFile
cbCmdFile

describe the character string containing the name of the command file to be opened. If *cbCmdFile* is 0, the user's default command file and [!Sys]<Sys>Cluster.cmds, the command file at the server, are opened.

Request Block

InitSysCmds is an object module procedure.

This page intentionally left blank

InitVidFrame (iFrame, iColStart, iLineStart, nCols, nLines, borderDesc, bBorderChar, bBorderAttr, fDblHigh, fDblWide): ercType

Description

InitVidFrame defines the screen coordinates and dimensions of one of the frames. InitVidFrame must be called at least once after the ResetVideo operation and before the InitCharMap operation is called. It can also be called while the video subsystem is in use to change a frame or to add a frame. The Video Control Block is updated to reflect the changed or added frame. (For the format of the frame descriptor and the Video Control Block, see Chapter 4, "System Structures.")

The screen coordinates of the upper left corner of the frame are specified by *iColStart* and *iLineStart*. The width and height of the frame are given by *nCols* and *nLines*, respectively. Frames can overlap, but they cannot exceed the screen dimensions.

Procedural Interface

InitVidFrame (iFrame, iColStart, iLineStart, nCols, nLines, borderDesc, bBorderChar, bBorderAttr, fDblHigh, fDblWide): ercType

where

iFrame

is an integer that ranges from 0 to the number of frame descriptors in the Video Control Block minus 1. This identifies the frame to be acted upon and selects one of the frame descriptors of the Video Control Block for modification. A value of 255 is treated as a special case and is reserved for internal use only.

iColStart

is the column of the screen that corresponds to the leftmost column of the frame.

iLineStart

is the line of the screen that corresponds to the top line of the frame.

nCols

is the width of the frame in columns.

nLines

is the height of the frame in lines.

borderDesc

is a byte with bits 0-3 specifying a border just outside the frame on the corresponding side. Note that the border characters are in addition to the area defined by *nCols* and *nLines*.

Bit	**Side**
0	Top
1	Right
2	Bottom
3	Left

The border is drawn when the InitCharMap operation is executed. The same character and attributes (*bBorderChar* and *bBorderAttr*) are used for all sides and corners.

bBorderChar

specifies the character code to use for the frame borders when drawn by the InitCharMap operation.

bBorderAttr

specifies the 4-bit character attribute field with which *bBorderChar* is to be displayed.

To create complex borders, including corner characters, initialize a frame that defines the entire screen; then put the appropriate border characters and attributes into the character map by using the PutFrameChars and PutFrameAttrs operations. (See the complete description of each operation in this chapter.)

fDblHigh

should be 0 except when this operation is used on B24 Teller Workstations. In this case, setting this flag to TRUE specifies the character attribute double high. *fDblWide* must also be set to TRUE.

fDblWide

should be 0 except when this operation is used on B24 Teller Workstations. In this case, setting this flag to TRUE specifies the character attribute double wide. *fDblHigh* must also be set to TRUE.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	10
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	75
12	iFrame	1	
13	iColStart	1	
14	iLineStart	1	
15	nCols	1	
16	nLines	1	
17	borderDesc	1	
18	bBorderChar	1	
19	bBorderAttr	1	
20	fDblHigh	1	
21	fDblWide	1	

InstallNet (iCase, pData, sData): ercType

Description

InstallNet passes CT-Net installation parameters to the operating system(s) of a server.

Procedural Interface

InstallNet (iCase, pData, sData): ercType

where

iCase

indicates the CT-Net parameters to be installed. Values and their descriptions are as follows:

Value	Parameter Description	Size (bytes)
0	Net Agent service exchange	2
1	Net Server remote user number range	4
2	local node name string:	
	string length (first byte)	1
	string	12 (max)
3	Net Server exchange	2

pData
sData

describes an area containing the CT-Net parameters. The value of *sData* must equal the size in bytes as required by *iCase*.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	1
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	344
12	iCase	2	
14	reserved	4	
18	pData	4	
22	sData	2	

KeyboardProfile (pIdRet, pwType)

Description

KeyboardProfile identifies the attached keyboard and returns its keyboard ID number. If KeyboardProfile cannot identify the keyboard, it returns a value of 9 at *pwType*.

Procedural Interface

KeyboardProfile (pbIdRet, pwType)

where

pIdRet

is the memory address of a byte where the ID of the attached keyboard is returned.

pwType

is the memory address of a word to which a value indicating the keyboard type is returned. Each value indicates the following:

Value	**Keyboard Type**
0	K1
1	K2
2	OEM special K4
3	K5
4	SG-101-K

Value	Keyboard Type
5	SG-102-K
6	Reserved for OEM
7	Reserved for OEM
8	ALC keyboard
9	Other keyboard

Request Block

KeyboardProfile is an object module procedure.

KillProcess (pid): ercType

Caution: *This operation is for* **restricted use only** *and is subject to change in future operating system releases.*

Description

The KillProcess primitive destroys the calling process. No resources other than the Process Control Block (PCB) are freed.

Procedural Interface

KillProcess (pid): ercType

where

pid

is the ID of the calling process.

Request Block

KillProcess is a Kernel primitive.

This page intentionally left blank

LaFromP (p, userNum, pPaRet): ercType

Description

LaFromP returns the 32-bit linear address corresponding to the logical memory address (p).

Procedural Interface

LaFromP (p, userNum, pPaRet): ercType

where

p

is the logical memory address.

userNum

is the user number returned from a CreatePartition or a GetPartitionHandle operation. If *userNum* is 0, then the user number of the calling program is used.

pPaRet

is the memory address to which the 32-bit linear address is returned.

Request Block

LaFromP is an object module procedure.

LaFromSn (sn, userNum, pPaRet): ercType

Caution: *This operation is supported by operating systems executing in protected mode only.*

Description

LaFromSn returns the linear address corresponding to the protected mode selector (SN).

Procedural Interface

LaFromSn (sn, userNum, pPaRet): ercType

where

sn

is the protected mode selector.

userNum

is the user number returned from a CreatePartition or a GetPartitionHandle operation. If *userNum* is 0, then the user number of the calling program is used.

pPaRet

is the memory address to which the linear address is returned.

Request Block

LaFromSn is a system-common procedure.

LibFree (lh): ercType

NOTE: This operation only works on virtual memory operating systems and is for use by programs executing in protected mode.

Description

LibFree notifies the operating system that the client no longer requires access to the specified dynamic link library (DLL). When the count of clients for a DLL reaches zero, the DLL is freed.

Procedural Interface

LibFree (lh): ercType

where

lh

is the handle (word) of the DLL to be freed.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	425
12	lh	2	

LibGetHandle (pbName, cbName, pLhRet): ercType

NOTE: This operation only works on virtual memory operating systems and is for use by programs executing in protected mode.

Description

LibGetHandle returns the dynamic link library (DLL) handle of the DLL with the specified name. If a library of the specified name is not loaded, LibGetHandle returns status code 1106 ("Bad lib name").

Applications may use this operation to determine if a DLL currently is loaded.

Procedural Interface

LibGetHandle (pbName, cbName, pLhRet): ercType

where

pbName
cbName

describe the DLL name.

pLhRet

is the memory address of the DLL handle returned.

Request Block

sLhRet is always 2.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	1
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	427
12	reserved	6	
18	pbName	4	
22	cbName	2	
24	pLhRet	4	
28	sLhRet	2	2

LibGetInfo (lh, tyInfo, pNameRet, sNameRet, pCbRet): ercType

NOTE: This operation only works on virtual memory operating systems and is for use by programs executing in protected mode.

Description

LibGetInfo returns the name or the file specification of the dynamic link library (DLL) specified by the DLL handle.

Procedural Interface

LibGetInfo (lh, tyInfo, pNameRet, sNameRet, pcbRet): ercType

where

lh

is the DLL handle (word).

tyInfo

is the type (word) of information requested. *tyInfo* is one of the following values:

Value	**Meaning**
0	Library name
1	Library file specification

pNameRet

is the memory address of the buffer to which the requested information is written.

sNameRet

is the size (word) of the buffer.

pCbRet

is the memory addess of the count of bytes returned.

Request Block

sCbRet is always 2.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	2
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	472
12	lh	2	
14	tyInfo	2	
16	reserved	2	
18	pInfoRet	4	
22	sInfoRet	2	
24	pCbRet	4	
28	sCbRet	2	2

LibGetProcInfo (lh, pbProcName, cbProcName, pProcInfoRet, sProcInfoRet): ercType

NOTE: This operation only works on virtual memory operating systems and is for use by programs executing in protected mode.

Description

LibGetProcInfo returns the memory address of the specified procedure in the specified dynamic link library (DLL).

Procedural Interface

LibGetProcInfo (lh, pbProcName, cbProcName, pProcInfoRet, sProcInfoRet): ercType

where

lh

is the DLL handle.

pbProcName
cbProcName

describe the name of the DLL procedure.

pProcInfoRet

is the memory address of the buffer to which the procedure information is written. The returned information has the following format:

Offset	**Field**	**Size (Bytes)**
0	ra	2
2	sn	2

sProcInfoRet

is the size of the buffer.

Request Block

Offset	**Field**	**Size (Bytes)**	**Contents**
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	1
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	428
12	lh	2	
16	reserved	4	
18	pbProcName	4	
22	cbProcName	2	
24	pProcInfoRet	4	
28	sProcInfoRet	2	

LibLoad (pbLibName, cbLibName, pLibInfoBlock, sLibInfoBlock, pFailNameRet, sFailNameRet, pCbFailNameRet, pLhRet): ercType

NOTE: This operation only works on virtual memory operating systems and is for use by programs executing in protected mode.

Description

LibLoad loads the dynamic link library (DLL) specified by *pbLibName/cbLibName* and returns a DLL handle. If loading the DLL causes the loading of one or more additional DLL modules and a module fails to load properly, the operation returns the name of the failed module at the caller-provided address.

Procedural Interface

LibLoad (pbLibName, cbLibName, pbLibInfoBlock, cbLibInfoBlock, pFailNameRet, sFailNameRet, pCbFailNameRet, pLhRet): ercType

where

pbLibName
cbLibName

describe the name of the DLL. If the name contains no suffix, the operating system appends *.dll*, for example the DLL name *PmWin* becomes *PmWin.dll*. The operating system uses the search path facility to access the file containing the DLL. (For details on DLL search paths, see "Setting Up DLL Search Paths" in the section entitled "Dynamic Link Libraries" in the *CTOS Operating System Concepts Manual*.)

pLibInfoBlock

is the memory address of the library information block. The structure has the following format:

Offset	**Field**	**Size (Bytes)**
0	wMinVersion	2
2	wMaxVersion	2

NOTE: Version 12.2 of the Module Definition Utility does not support the setting of versions in DLLs, as defined by pLibInfoBlock. See the CTOS Programming Utilities Reference Manual: Building Applications.

sLibInfoBlock

is the size (word) of the library information block.

pFailNameRet

is the memory address of the name of the DLL causing the load to fail.

sFailNameRet

is the size (word) of the DLL name string.

pCbFailNameRet

is the memory address of the count of bytes returned.

pLhRet

is the memory address of the DLL handle returned.

Request Block

sCbFailNameRet and *sLhRet* are always 2.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	2
3	nRespPbCb	1	3
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	408
12	reserved	6	
18	pbLibName	4	
	cbLibName	2	
24	pLibInfoBlock	4	
28	sLibInfoBlock	2	
30	pFailNameRet	4	
34	sFailNameRet	2	
36	pCbFailNameRet	4	
40	sCbFailNameRet	2	2
42	pLhRet	4	
46	sLhRet	2	2

This page intentionally left blank.

LoadBackgroundPalette (pAttrs, sAttrs): ercType

NOTE: This operation is supported by protected mode operating systems only.

Description

LoadBackgroundPalette specifies an array of bytes to be used as a color palette for background colors. The definition of these byte values is identical to the definition for foreground colors, and is described in "Using Color" in the *CTOS Programming Guide*.

Status code 7 ("Not implemented") is returned if the workstation does not support the display of background color.

Procedural Interface

LoadBackgroundPalette (pAttrs, sAttrs): ercType

where

pAttrs

is the memory address of the array of bytes to be used as the background color palette.

sAttrs

is the size (word) of the palette. *sAttrs* must be 8 bytes or less. If *sAttrs* is a value less than 8, only the number of bytes provided by *sAttrs* are overwritten. Normally, however, all 8 bytes are overwritten.

Request Block

LoadBackgroundPalette is an object module procedure.

This page intentionally left blank.

LoadColorStyleRam (pAttrs): ercType

Caution: *This operation is documented for compatibility with older programs. New programs should use ProgramColorMapper.*

Description

LoadColorStyleRam specifies eight bytes that are passed to the color graphics style RAM. These attribute settings are used to display different combinations of color, reverse video, and underlining on different sections of the video screen. The low-order six bits of each byte specify the color and intensity, and the high-order two bits are used for reverse-video and underlining, respectively.

Procedural Interface

LoadColorStyleRam (pAttrs): ercType

where

pAttrs

points to the memory address of the 8-byte set of attributes.

Request Block

LoadColorStyleRam is an object module procedure.

This page intentionally left blank

LoadFontRam (fh, pBuffer, sBuffer): ercType

NOTE: The direct, memory, and pointer options are not available on real mode operating systems. LoadFontRam calls with pointer mode are not supported under Context Manager.

Description

LoadFontRam prepares subsequent character output to use the specified font. The modes of transfer are determined by the values of *fh, pBuffer,* and *sBuffer* as follows:

mode	**fh**	**pBuffer**	**sBuffer**
normal	non-zero	pointer	non-zero
direct	non-zero	not used	0
memory	0	pointer	non-zero
pointer	0	pointer	0
clear font	0	0	0

Caution: *Normal is the only mode that can be used on real mode operating systems. Use of direct, memory, or pointer mode can have unpredictable results.*

The effect of LoadFontRam in each mode is as follows:

Mode	**Effect**
normal	reads a font from the specified open file using the specified work area and transfers the font to the font RAM. This mode works on all operating systems.

direct	reads a font from a specified open file and transfers the font directly to the font RAM.
memory	reads a font from memory and transfers that font to the font RAM.
pointer	replaces the system font pointer with the memory address of the specified work area.
clear font	indicates that the current font should be replaced by the system font during the next ResetVideo/ResetVideoGraphics operation.

Bit Map Devices

For bit map devices, such as the GC003, see Table 3-3 in the *Graphics Programmer's Guide*. The table provides details on the bit map font format.

Character Map Devices

For character-map devices (for example, the character NGEN), the file must contain a 16-word entry for each of 256 characters. Thus the file is exactly 4096 words (8192 bytes) long.

For workstations, the format of the data in the font file and the font RAM is the same. Words 12 through 15 of each 16-word entry must be zero. Words 0 through 11 represent the twelve rows of the character from top to bottom: In each of these words the bits are defined as follows:

Bits	**Meaning**
8 through 0	Represent the pixels from left to right. Bit 0 is the least significant.
9	Enables half-pixel shift of the pixel row.
10 through 15	Should be 0 in the font file. These bits are not actually present in the font RAM, which is implemented as an array of 4096 10-bit words (aligned on 16-bit word boundaries in the address space of an 80186 processor).

Procedural Interface

LoadFontRam (fh, pBuffer, sBuffer): ercType

where

fh

is the file handle of an open file containing the font.

pBuffer

is the memory address of the work area to be used in loading the font RAM.

sBuffer

describes the work area. *sBuffer* may be any size; however, in normal mode, the buffer will be truncated to the nearest multiple of 512 bytes.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	1
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	22
12	fh	2	
14	reserved	4	
18	pBuffer	4	
22	sBuffer	2	

LoadInteractiveTask (userNum, pbFileSpec, cbFileSpec, pbPassword, cbPassword, priority, fDebug): ercType

Description

LoadInteractiveTask is the same as LoadPrimaryTask, with the addition of the *fDebug* parameter. (See LoadPrimaryTask.)

The process priority specified by LoadInteractiveTask may be modified by SetDeltaPriority before the process is scheduled for execution. (See SetDeltaPriority.)

Procedural Interface

LoadInteractiveTask (userNum, pbFileSpec, cbFileSpec, pbPassword, cbPassword, priority, fDebug): ercType

where

userNum

is the user number returned from a CreatePartition or GetPartitionHandle operation.

pbFileSpec
cbFileSpec

describe a character string that is the name of the executable file to be loaded.

pbPassword
cbPassword

describe a password that authorizes access to the specified file.

priority

is the initial priority of the process that is created.

fDebug

indicates whether or not the run file is to be debugged. TRUE indicates that it will not be scheduled for execution and that the Debugger is entered as soon as the program is loaded into the partition.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	2
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	262
12	userNum	2	
14	priority	2	
16	fDebug	2	
18	pbFileSpec	4	
22	cbFileSpec	2	
24	pbPassword	4	
28	cbPassword	2	

LoadPrimaryTask (userNum, pbFileSpec, cbFileSpec, pbPassWord, cbPassWord, priority): ercType

Description

LoadPrimaryTask loads and activates the run file specified by the file specification in the vacant application partition.

The process priority specified by LoadPrimaryTask may be modified by SetDeltaPriority before the process is scheduled for execution. (See SetDeltaPriority.)

LoadPrimaryTask performs the following functions:

1. Verifies that the file specification specifies a run file that contains a valid segment arrangement and that the run file fits in the application partition.

2. Allocates a short-lived memory segment large enough to contain the specified run file.

3. Reads the run file into the application partition.

4. Relocates all intersegment references to accommodate the memory address at which the run file is loaded.

5. Creates a process to be scheduled at the specified priority. The initial values loaded into the segment registers (Code Segment (CS), Data Segment (DS), Stack Segment (SS), Extra Segment (ES)), the Stack Pointer (SP), and the Instruction Pointer (IP) are derived from information in the run-file header.

Procedural Interface

LoadPrimaryTask (userNum, pbFileSpec, cbFileSpec, pbPassWord, cbPassWord, priority): ercType

where

userNum

is the user number returned from a CreatePartition or a GetPartition-Handle operation.

pbFileSpec
cbFileSpec

describe a character string of the following form: {Node}[VolName]<DirName>FileName. The distinction between uppercase and lowercase in file specifications is not significant in matching file names.

pbPassWord
cbPassWord

describe the volume, directory, or file password that authorizes access to the specified file.

priority

is the priority (0-254, with 0 the highest) at which to schedule the newly created process for execution.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	2
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	178
12	userNum	2	
14	priority	2	
16	reserved	2	
18	pbFileSpec	4	
22	cbFileSpec	2	
24	pbPassWord	4	
28	cbPassWord	2	

This page intentionally left blank

LoadRunFile (fh, priority, fSuspend, pIdRet, pPdbRet, sPdbMax): ercType

Description

LoadRunFile obtains the code and/or data from the specified run file on disk and places it in linear memory in executable form. The operation returns a Process Descriptor Block (PDB). Using information contained in the PDB, the caller can either

- call CreateProcess to create a new process thread
- use the CS:IP in the PDB to call the code from an existing thread

Later, the code or data can be deallocated using DeallocRunFile. Relative to allocation, deallocations must occur in the exact opposite sequence. (For details, see DeallocRunFile.)

Procedural Interface

LoadRunFile (fh, priority, fSuspend, pIdRet, pPdbRet, sPdbMax): ercType

where

fh

is the file handle of the run file that has been opened by the caller.

priority

is the priority (255). Any other value returns error code 249.

fSuspend

is a flag that indicates whether or not the process (if created) is to be scheduled for execution. TRUE means the process is created but its state is suspended. FALSE means the process is scheduled for execution based upon its priority.

pIdRet

is the memory address of an area into which the 4 byte identification number for the run file is stored. This information is required to make subsequent calls to DeallocRunFile.

pPdbRet

is the memory address of an area into which the Process Descriptor Block will be placed. (See CreateProcess for the format of the returned data.)

sPdbMax

is the maximum size in bytes of the Process Descriptor Block.

Request Block

sIdRet is always 4.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	2
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	350
12	fh	2	
14	priority	2	
16	fSuspend	2	
18	pIdRet	4	
22	sIdRet	2	4
24	pPdbRet	4	
28	sPdbMax	2	

This page intentionally left blank.

LoadRunFile (fh, priority, fSuspend, pIdRet, pPdbRet, sPdbMax): ercType

Description

LoadRunFile allocates code or data and loads it into memory. Later, the code or data can be deallocated using DeallocRunFile. (See DeallocRunFile.)

Procedural Interface

LoadRunFile (fh, priority, fSuspend, pIdRet, pPdbRet, sPdbMax): ercType

where

fh

is the file handle of the run file that has been opened by the caller.

priority

is the priority (0-254, with 0 the highest) at which to schedule the newly created process for execution. A value of 255 requests that a process not be created. This permits loading of a run file that is executed by calling the operations in it from another process.

fSuspend

is a flag that indicates whether or not the process (if created) is to be scheduled for execution. TRUE means the process is created but its state is suspended. FALSE means the process is scheduled for execution based upon its priority.

pIdRet

is the memory address of an area into which the 4 byte identification number for the run file is stored. This information is required to make subsequent calls to DeallocRunFile.

pPdbRet

is the memory address of an area into which the Process Descriptor Block will be placed. (See CreateProcess for the format of the returned data.)

sPdbMax

is the maximum size in bytes of the Process Descriptor Block.

Request Block

sIdRet is always 4.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	2
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	350
12	fh	2	
14	priority	2	
16	fSuspend	2	
18	pIdRet	4	
22	sIdRet	2	4
24	pPdbRet	4	
28	sPdbMax	2	

LoadTask (fh, priority, fDebug): ercType

Description

LoadTask loads and activates an additional run file (secondary task) as part of the current application program in the application partition. (See "Partitions and Partition Management" in the *CTOS Operating System Concepts Manual* for more information on application partitions with more than one run file.)

The process priority specified by LoadTask may be modified by SetDeltaPriority before the process is scheduled for execution. (See SetDeltaPriority.)

LoadTask performs the following functions:

1. Verifies that the file handle specifies a run file that contains a valid segment arrangement and that the run file fits in the application partition memory.
2. Allocates a short-lived memory segment large enough to contain the specified run file.
3. Reads the run file into the application partition.
4. Relocates all intersegment references to accommodate the memory address at which the run file is loaded.
5. Creates a process to be scheduled at the specified priority. The initial values loaded into the segment registers (Code Segment (CS), Data Segment (DS), Stack Segment (SS), Extra Segment (ES)), the Stack Pointer (SP), and the Instruction Pointer (IP) are derived from information in the run-file header.

Procedural Interface

LoadTask (fh, priority, fDebug): ercType

where

fh

is the file handle of a run file that has been opened by the calling process.

priority

is the priority (0-254, with 0 the highest) at which to schedule the newly created process for execution. A value of 255 requests that a process not be created. This permits the loading of a run file that is executed by calling the operations in it from another process.

fDebug

indicates whether the program is to be debugged. TRUE indicates it is to be debugged and therefore not scheduled for execution; FALSE indicates it is to be scheduled for execution. In contrast to its meaning in the Chain operation, setting *fDebug* to TRUE does *not* automatically activate the Debugger.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb1	0	
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	29
12	fh	2	
14	priority	2	
16	fDebug	2	

This page intentionally left blank

LockIn (wPort): bValueRet

Description

LockIn allows you to read from an I/O port. The use of this operation is essential on certain types of workstation hardware because of the timing functions it performs.

Historically, the LockIn and LockOut operations were designed to program the serial communications controller at the interface level closest to the hardware. (See the description of LockOut.) The operations, however, are not restricted to serial ports.

Procedural Interface

LockIn (wPort): bValueRet

where

wPort

is the input/output port from which a byte value is to be read.

bValueRet

is the byte value read.

Request Block

LockIn is an object module procedure.

This page intentionally left blank

LockInContext (userNum): ercType

NOTE: On virtual memory operating systems, this operation does not lock the user's pages in memory.

Description

LockInContext causes the specified user number (partition) to be made memory-resident and unswappable (locked-in). Specifying a user number of 0 causes the caller's partition to be locked-in. A lock-in asserted using LockInContext may be removed using the UnlockInContext operation.

Procedural Interface

LockInContext (userNum): ercType

where

userNum

specifies the user number (word) to be locked-in.

Request Block

LockInContext is a system-common procedure.

This page intentionally left blank.

LockOut (wPort, bValue)

Description

LockOut allows you to write to the serial input/output port. The use of this operation is essential on certain types of workstation hardware because of the timing functions it performs.

Historically, the LockOut and LockIn operations were designed to program the serial communications controller at the interface level closest to the hardware. (See the description of LockIn.) The operations, however, are not restricted to serial ports.

Procedural Interface

LockOut (wPort, bValue)

where

wPort

  is the input/output port where a byte value is to be written.

bValue

  is the byte value to be written.

Request Block

LockOut is an object module procedure.

This page intentionally left blank

LockVideo

Caution: *For compatibility on all workstations, it is recommended that you use the Video Access Method (VAM) rather than using this operation. (See "Video" in the CTOS Operating System Concepts Manual for more information.)*

Description

LockVideo is used to lock the video structures used by the operating system. It is used by programs that read or write directly to the screen at an interface level below the Video Access Method (VAM) operations.

A program that writes directly to the screen should use GetPStructure to get the memory address of the partition's *rgpVidMemLine* structure. This is an array of memory addresses of each line of the video refresh buffer, which includes both characters and attributes, as described in the hardware manuals for each type of workstation.

There are as many addresses as there are lines on the particular workstation. (You can obtain the number of lines for your workstation by calling the QueryVidHdw or the QueryVideo operation.)

The addresses in *rgpVidMemLine* are changed by the Context Manager whenever the ownership of the video is changed. The refresh buffer pointing to them is not necessarily contiguous.

Whenever the program wishes to check or update the refresh buffer, a call to LockVideo should be made, followed by the use of *rgpVidMemLine,* followed by a call to UnlockVideo. The LockVideo call will always return unless another process has made a call to LockVideoForModify.

LockVideo uses an exchange defined at system build for synchronization. If an erroneous message comes from the exchange, the calling program to LockVideo is terminated with status code 509, ("Wrong message").

Procedural Interface

LockVideo

Request Block

LockVideo is a system–common procedure.

LockVideoForModify

Caution: *It is recommended that you use the Video Access Method (VAM) rather than using this operation for compatibility on all workstations. (See "Video" in the CTOS Operating System Concepts Manual for more information.)*

Description

LockVideoForModify should not be used by programs that run under a partition-managing program, such as the Context Manager.

This operation is used by programs that want to modify the video structures used by the operating system. After this operation is called, any process that calls a Video Access Method (VAM) operation or calls LockVideo will be suspended until UnlockVideoForModify is called.

The process calling LockVideoForModify will be suspended while there are any processes that have already made a call to LockVideo and have not made a call to UnlockVideo.

LockVideoForModify uses an exchange defined at system build for synchronization. If an erroneous message comes to the exchange, a system crash occurs with status code 509 ("Wrong lock message").

Procedural Interface

LockVideoForModify

Request Block

LockVideoForModify is a system–common procedure.

This page intentionally left blank

LockXbis (ppXbisRet): ercType

Description

LockXbis is used to reserve the X-Bus Information Structure (XBIS) at memory location 400h. (See "X-Bus Management" in the *CTOS Operating System Concepts Manual.*)

The operation returns a pointer to location 400h in anticipation of future products that have memory mapping and protection.

Procedural Interface

LockXbis (ppXbisRet): ercType

where

ppXbisRet

describes the memory area to which the memory address of the XBIS is returned.

Request Block

spXbisRet is always 4.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	0
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	32797*
12	ppXbisRet	4	
16	spXbisRet	2	4

*On BTOS II, 3.2 workstation operating systems, the request code 8202 is also supported.

LogMsgIn: ercType

Description

LogMsgIn logs the current client's message pointed to by the global variable *pRq*. LogMsgIn is used by an asynchronous system service when it receives a message from a client at its exchange.

Procedural Interface

LogMsgIn: ercType

Request Block

LogMsgIn is an object module procedure in Async.lib.

This page intentionally left blank

LogRespond(pRqBlock): ercType

Description

LogRespond logs the response to the current client's request pointed to by the global variable *pRq*. LogRespond is used by an asynchronous system service before it responds to the client request. It is also used by the asynchronous system service library procedures SwapContextUser and TerminateContextUser before a response is made to outstanding client requests.

Procedural Interface

LogRespond(pRqBlock): ercType

where

pRqBlock

is the memory address of the client's request block.

Request Block

LogRespond is an object module procedure in Async.lib.

This page intentionally left blank

LogRequest: ercType

Description

LogRequest logs the current client's request pointed to by the global variable *pRq*. LogRequest is called by the asynchronous system service procedures BuildAsyncRequest and BuildAsyncRequestDirect before Request and RequestDirect, respectively, are called.

Procedural Interface

LogRequest: ercType

Request Block

LogRequest is an object module procedure in Async.lib.

This page intentionally left blank

LookUpField (fh, pBuffer, sBuffer, pbFieldName, cbFieldNameMax, pcbFieldName, pbValue, cbValueMax, pcbValueRet, fAllowWhiteSpace): ercType

Description

LookUpField reads from a file and searches for a ":FieldName:" string. The search begins at the current location in the file. The operation returns to the caller both the :FieldName: string and the string value found after the trailing colon. Only one file can be scanned at a time.

Procedural Interface

LookUpField (fh, pBuffer, sBuffer, pbFieldName, cbFieldNameMax, pcbFieldName, pbValue, cbValueMax, pcbValueRet, fAllowWhiteSpace): ercType

where

fh

is the file handle of the file to be searched.

pBuffer
sBuffer

describe a buffer to be used in the search. *sBuffer* must be a multiple of 512.

pbFieldName
cbFieldNameMax

describe the memory area to which the *FieldName* string is to be copied. The colons are removed from the *FieldName* string before it is copied.

pcbFieldName

is the memory address of the word to which the count of bytes in the FieldName string is returned.

pbValue
cbValueMax

describe the memory area to which the Value string is to be copied.

pcbValueRet

is the memory address of a word to which the count of bytes in the Value string is returned.

fAllowWhiteSpace

is TRUE or FALSE. White space characters are space, tab, linefeed, and carriage return.

Request Block

LookUpField is an object module procedure.

LookUpNumber (fh, pBuffer, sBuffer, pbFieldName, cbFieldName, pwRet): ercType

Description

LookUpNumber reads from a file, searches for a ':FieldName:' string, and returns the string length to the caller. The search begins at the current location in the file. What follows the string is interpreted as a decimal number and is returned to the caller. Only one file can be scanned at a time.

Procedural Interface

LookUpNumber (fh, pBuffer, sBuffer, pbFieldName, cbFieldName, pwRet): ercType

where

fh

is the file handle of the file to be searched.

pBuffer
sBuffer

describe a buffer to be used in the search. *sBuffer* must be a multiple of 512.

pbFieldName
cbFieldName

describe the string to be searched for. The colons should not be given as part of the field name.

pwRet

is the memory address of a two-byte area where the numeric value is returned.

Request Block

LookUpNumber is an object module procedure.

LookUpReset: ercType

Description

LookUpReset resets the point from which a scan begins to the beginning of the current file. Only one file can be scanned and, therefore, reset at a time.

LookUpReset is used in conjunction with LookUpField, LookUpNumber, and LookUpString to parse configuration files.

Procedural Interface

LookUpReset: ercType

Request Block

LookUpReset is an object module procedure.

This page intentionally left blank

LookUpString (fh, pBuffer, sBuffer, pbFieldName, cbFieldName, pbValue, cbValueMax, pcbValueRet): ercType

Description

LookUpString reads from a file and searches for a ':FieldName:' string. The search begins at the current location in the file. The string value found after the trailing colon is returned to the caller. Only one file can be scanned at a time.

Procedural Interface

LookUpString (fh, pBuffer, sBuffer, pbFieldName, cbFieldName, pbValue, cbValueMax, pcbValueRet): ercType

where

fh

is the file handle of the file to be searched.

pBuffer
sBuffer

describe a buffer to be used in the search. *sBuffer* must be a multiple of 512.

pbFieldName
cbFieldName

describe the string to be searched for. The colons should not be given as part of the field name.

pbValue
cbValueMax

describe the memory area where the string found is to be copied.

pcbValueRet

is the memory address of a two-byte area where the count of bytes in the string is returned.

Request Block

LookUpString is an object module procedure.

LookUpValue (fh, pBuffer, sBuffer, pbFieldName, cbFieldName, pRet, sRet): ercType

Description

LookUpValue reads from a file, searches for a ':FieldName:' string, and returns the numeric value of the following string to the caller. The search begins at the current location in the file. What follows the ':FieldName:' string is interpreted as a decimal or hexadecimal number, which is decoded and returned (See LookUpNumber) to the caller.

Procedural Interface

LookUpValue (fh, pBuffer, sBuffer, pbFieldName, cbFieldName, pRet, sRet): ercType

where

fh

is the file handle of the file to be searched.

pBuffer
sBuffer

describe a buffer to be used in the search. *sBuffer* must be a multiple of 512.

pbFieldName
cbFieldName

describe the string to be searched for. The colons should not be given as part of the field name.

pRet
sRet

describe the memory area of where the value is returned. A byte, word, or quad can be returned, depending on the value of sRet.

sRet	**Description**
1	Byte (8-bit)
2	Word (16-bit)
4	Quad (32-bit)

Request Block

LookUpValue is an object module procedure.

MakePermanent: ercType

Description

MakePermanent makes the overlay from which it is called permanently resident in memory in a program using the Virtual Code facility. The overlay remains resident until it is released with a call to ReleasePermanence.

MakePermanent is similar to MakePermanentP, but the latter operation allows you to make an arbitrary overlay permanent. (See MakePermanentP.)

NOTE: Using MakePermanent reduces the effective size of the overlay zone and may cause performance problems if the resulting size is too small.

Procedural Interface

MakePermanent: ercType

Request Block

MakePermanent is an object module procedure.

This page intentionally left blank

MakePermanentP (pProcInfo): ercType

Description

MakePermanentP makes an arbitrary overlay permanently resident in memory in a program using the Virtual Code facility. The overlay is resident until it is released with a call to ReleasePermanence.

To use this operation, you must provide the memory address of the specified overlay. You can obtain this address by calling MapIOvlyCs. (See MapIOvlyCs.)

NOTE: Using MakePermanentP reduces the effective size of the overlay zone and may cause performance problems if the resulting size is too small.

Procedural Interface

MakePermanentP (pProcInfo): ercType

where

pProcInfo

is the memory address of the overlay to be made permanent. (The *iOvRet* parameter of the MapCsIOvly operation can be used to compute *pProcInfo* prior to calling this operation.)

Request Block

MakePermanentP is an object module procedure.

This page intentionally left blank

MakeRecentlyUsed

NOTE: This operation does not return a status code.

Description

MakeRecentlyUsed adjusts the apparent age of an the overlay from which it is called, making the overlay less likely to be swapped out of memory in a program using the Virtual Code facility.

The default replacement algorithm of the Virtual Code facility swaps out overlays based on age in memory. MakeRecentlyUsed overrides this default by making an overlay appear to have zero age regardless of when it was swapped in.

When an overlay calls MakeRecentlyUsed, the overlay is replaced in memory only if there is insufficient room for it and for the next overlay called.

MakeRecentlyUsed and UpdateOverlayLru are similar in that they override the Virtual Code facility's default replacement algorithm. Neither operation, however, guarantees that the overlay will remain in memory permanently. Use MakePermanent or MakePermanentP if your intent is to keep an overlay permanently resident in memory.

UpdateOverlayLru differs from MakeRecentlyUsed in preventing any other overlay (not the calling overlay) from being swapped out. (See UpdateOverlayLru.)

Procedural Interface

MakeRecentlyUsed

Request Block

MakeRecentlyUsed is an object module procedure.

*MapBusAddress (pb, cb, wDmaBoundaryType, pBusAddressRet,
pCbMappedRet): ercType*

NOTE: This operation is supported by protected mode operating systems only.

Description

MapBusAddress converts a segmented address (in SN:RA form) to a bus address and, on some systems, programs hardware necessary for the bus address to generate a reference to the appropriate physical address. (For a description of bus addresses, see "Bus Address Management" in the *CTOS Operating System Concepts Manual.*)

Once the bus address is no longer required, the complementary operation, UnmapBusAddress, should be used to free any hardware resources that were allocated by MapBusAddress.

Procedural Interface

*MapBusAddress (pb, cb, wDmaBoundaryType, pBusAddressRet,
pCbMappedRet): ercType*

where

pb
cb

describe an area in memory that is to be referenced by the returned bus address. Both the beginning logical address of the memory and the length of the memory to be referenced must be specified.

wDmaBoundaryType

is a word value that indicates the DMA hardware limitations relevant to the device for which the bus address is requested. Permissible values are

0	noDmaBoundary: No DMA boundary needs to be taken into consideration.
1	byteDmaBoundary: DMA boundaries occur at every 64K bytes of physical memory. For example, a DMA transfer could not cross from 1FFFFh to 20000h without a pause to reprogram the DMA hardware.
2	wordDmaBoundary: DMA boundaries occur at every 64K words of physical memory. For example, a DMA transfer could not cross from 3FFFEh to 40000h without a pause to reprogram the DMA hardware.

SRP processors (including the 80386-based GP family) do not have any DMA boundaries. Dual floppy modules and floppy/hard disk modules for workstations have DMA boundaries that occur at 64 kilobyte boundaries in physical memory, workstation hard disk upgrade and expansion modules have DMA boundaries that occur at 64 kiloword boundaries in physical memory while internal SCSI disks in the Series *i* workstations and SCSI upgrade modules do not have DMA boundaries.

pBusAddressRet

is the address of a double word to which the calculated bus address is returned.

pCbMappedRet

is the address of a word to which the actual count of bytes that may be transferred to or from the bus address is returned. Because of DMA boundary limitations, this may be less than the count that was requested and may even be 0.

Request Block

MapBusAddress is a system-common procedure.

This page intentionally left blank.

MapCsIOvly(cs, pIOvRet): ercType

Description

MapCsIOvly is a Virtual Code utility that takes the code segment (CS) part of a memory address and returns the overlay in which that address is currently contained.

For example, the value of *iOvRet* could be used to compute the overlay's address *(pProcInfo)* prior to a call to MakePermanentP. (See MakePermanentP.)

Procedural Interface

MapCsIOvly(cs, pIOvRet): ercType

where

cs

is the code segment portion of the memory address about which the inquiry is being made.

pIOvRet

is the memory address to which the index of the overlay containing that cs is returned.

Request Block

MapCsIOvly is an object module procedure.

This page intentionally left blank

MapIOvlyCs (IOv, pCsRet): ercType

Description

MapIOvlyCs is a Virtual Code utility that takes the index of an overlay and returns the code segment (CS) part of an overlay's current memory address.

You can use this operation to obtain an overlay's address before calling MakePermanentP or any other operation that requires the address as a parameter.

For example, you would combine an initial pointer (IP) value of 0 with the CS returned by this operation to generate the overlay's memory address. You would pass the memory address to MakePermanentP to make the overlay permanently resident in memory.

Procedural Interface

MapIOvlyCs (IOv, pCsRet): ercType

where

IOv

is the index of the overlay about which the inquiry is being made.

pCsRet

is the memory address of the location where the CS is returned.

Request Block

MapIOvlyCs is an object module procedure.

This page intentionally left blank

MapPhysicalAddress (qPhysAddr, qSizeOfRange, fReturnSelector, pAddrRet): ercType

NOTE: This operation only works on virtual memory operating systems and is for use only by programs executing in protected mode.

Description

MapPhysicalAddress returns the linear address/selector that corresponds to the specified physical address in nonprocessor memory (for example, a SuperGen X-Bus+ cartridge). If the calling program supplies a range of physical addresses, MapPhysicalAddress returns either the selector for the range or the linear address of the first memory location in the range. The complementary operation, UnmapPhysicalAddress, deallocates the selector or linear addresses. (See UnmapPhysicalAddress.)

MapPhysicalAddress is useful when the program must access a particular physical address in nonprocessor memory. A program cannot directly reference a physical address; rather, it must specify a linear address. MapPhysicalAddress provides the program with a linear address that can be used to reference the physical address. As an example, the Audio Service uses MapPhysicalAddress to determine the linear addresses of memory locations on the Audio/Video cartridge.

MapPhysicalAddress also disables caching of the specified physical addresses. This feature allows the program to directly access nonprocessor memory without interference from the memory cache.

If the application provides a physical address from processor memory, status code 435 ("Bad physical address") is returned.

Procedural Interface

MapPhysicalAddress (qPhysAddr, qSizeOfRange, fReturnSelector, pAddrRet): ercType

where

qPhysAddr

is the first address in the range of physical addresses for which the selector or linear address is to be returned.

qSizeOfRange

is the number of bytes in the physical address range.

fReturnSelector

is a flag that, when TRUE, causes MapPhysicalAddress to return a selector that addresses the physical address range.. If FALSE, this flag causes MapPhysicalAddress to return the linear address of the first memory location in the physical address range.

pAddrRet

is the memory address of a double word to which the linear address or selector is returned. A selector is always returned in the high-order word.

Request Block

sAddrRet is always 4.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	10
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	467
12	fReturnSelector	2	
14	qPhysAddr	4	
18	qSizeOfRange	4	
22	pAddrRet	4	
26	sAddrRet	2	4

This page intentionally left blank

MapPStubPProc (pStub, ppProcRet): ercType

Caution: *In protected mode, this operation returns status code 7 ("Not implemented").*

Description

MapPStubPProc of the Virtual Code facility returns the real address of a procedure (*pProc* portion of a stub, which is the last 4 bytes) without your having to know the internal structure of the stub.

Procedural Interface

MapPStubPProc (pStub, ppProcRet): ercType

where

pStub

is the memory address of the stub.

ppProcRet

is the memory address to which the real address of the procedure (*pProc*) is to be placed.

Request Block

MapPStubPProc is an object module procedure.

This page intentionally left blank

MapXBusWindow (bType, bModule, ppWindowRet, sWindow, pEarRet, oWindowPort): ercType

Caution: *This operation works on X-Bus workstation hardware only. Do not use it on Shared resource processor or SuperGen Series 5000 hardware.*

Description

MapXBusWindow sets up the X-Bus Extended Address Register (EAR) to access the memory within an X-Bus module. This operation is similar to MapXBusWindowLarge. It is recommended that you use MapXBusWindowLarge instead of this operation in all new programs. (See MapXBusWindowLarge.)

MapXBusWindow is documented for compatibility with old programs only. If you continue to use MapXBusWindow to perform pointer arithmentic on the X-Bus window memory address (*pWindowRet*) returned, your program will not work in protected mode.

Procedural Interface

MapXBusWindow (bType, bModule, ppWindowRet, sWindow, pEarRet, oWindowPort): ercType

where

bType

is the module type code.

bModule

is the module position index (of all modules of the specified type). For example, 0 refers to the leftmost module of the type *bType*.

ppWindowRet

is the memory address of a doubleword where the start address of the assigned window is returned.

sWindow

is the size of the window desired (in K bytes). It must be less than or equal to that defined in the System Configuration File.

pEarRet

is the memory address of a word where the value written to the X-Bus Extended Address Register (EAR) is copied.

oWindowPort

is the offset within the module's I/O space of its window mapping register.

Request Block

spWindowRet is always 4. *sEarRet* is always 2.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	2
4	userNum	2	
6	exchRet	2	
8	ercRet	2	
10	rqCode	2	264
12	bType	1	
13	bModule	1	
14	sWindow	2	
16	oWindowPort	2	
18	ppWindowRet	4	
22	spWindowRet	2	4
24	pEarRet	4	
28	sEarRet	2	2

This page intentionally left blank

MapXBusWindowLarge (bType, bModule, prgpWindowRet, srgpWindowRet, pipWindowRetMac, sWindow, pEarRet, oWindowPort): ercType

Caution: *This operation works on X-Bus workstation hardware only. Do not use it on Shared resource processor or SuperGen Series 5000 hardware.*

Description

MapXBusWindowLarge sets up the X-Bus Extended Address Register (EAR) to access the memory within an X-Bus module. The X-Bus EAR takes an address that is generated in an application program and maps it to an address that is understood by the X-Bus module.

Since there is only one X-Bus EAR, the EAR is maintained as part of the context of the process. The operating system saves and restores it automatically. It is, however, returned in *pEarRet*. This value is needed by the application program only if the X-Bus memory is to be accessed by interrupt routines. The SwapXBusEar system-common procedure is provided for this purpose.

MapXBusWindowLarge must be issued at least once before an application program attempts to access a memory-mapped X-Bus slave module and must be issued again if the program wishes to access a different module. It may be issued as many times as desired if the program wishes to reaccess a module after accessing another; the only restriction is that *oWindowPort* must be the same for all requests accessing the same module.

Procedural Interface

MapXBusWindowLarge (bType, bModule, prgpWindowRet, srgpWindowRet, pipWindowRetMac, sWindow, pEarRet, oWindowPort): ercType

where

bType

is the module type code.

bModule

is the module position index (of all modules of the specified type). For example, 0 refers to the leftmost module of the type *bType*.

prgpWindowRet

is the address of an area within the caller's memory, which MapXBusWindowLarge fills in with an array of addresses. The first array element is the starting address of a 64K-byte memory block reserved for the X-Bus window. Additional array elements, if any, are the starting addresses of additional memory blocks reserved for the window. Each starting address is 64K bytes from the last.

srgpWindowRet

is the size in bytes of the memory area supplied.

pipWindowRetMax

is the memory address of the number of array elements defined. The number of elements returned will be 2, 4, or 8.

sWindow

is the size of the window desired (in K bytes). It must be less than or equal to that defined in the System Configuration File.

pEarRet

is the memory address of a word where the value written to the X-Bus Extended Address Register (EAR) is copied.

oWindowPort

is the offset within the module's I/O space of its window mapping register.

Request Block

sEarRet and *sipWindowRetMax* are always 2.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	3
4	userNum	2	
6	exchRet	2	
8	ercRet	2	
10	rqCode	2	321
12	bType	1	
13	bModule	1	
14	sWindow	2	
16	oWindowPort	2	
18	prgpWindowRet	4	
22	srgpWindowRet	2	
24	pEarRet	4	
28	sEarRet	2	2
30	pipWindowRetMax	4	
34	sipWindowRetMax	2	2

This page intentionally left blank

MarkKeyedQueueEntry (pbQueueName, cbQueueName, pbKey1, cbKey1, oKey1, pbKey2, cbKey2, oKey2, pEntryRet, sEntryRet, pStatusBlock, sStatusBlock): ercType

Description

MarkKeyedQueueEntry is used by the queue server to obtain the first unmarked queue entry with up to two key fields equal to the values specified. MarkKeyedQueueEntry marks the queue entry as being in use, reads it into a buffer, and then returns a queue entry handle by which the queue entry is identified in a subsequent call to RemoveMarkedQueueEntry.

The byte count of at least one key field (either *cbKey1* or *cbKey2*) must be nonzero. If one key field is nonzero, that field only is used in the search. If both are nonzero, both are used in the search.

Each nonzero key field must match a specified string in the queue entry.

Procedural Interface

MarkKeyedQueueEntry (pbQueueName, cbQueueName, pbKey1, cbKey1, oKey1, pbKey2, cbKey2, oKey2, pEntryRet, sEntryRet, pStatusBlock, sStatusBlock): ercType

where

pbQueueName
cbQueueName

describe a queue name corresponding to a queue name specified when the queue was installed.

pbKey1
cbKey1

describe a key field to be compared with a string located at an offset *oKey1* in the queue entry.

oKey1

is the offset of the string key field in the queue entry. The offset starts from byte 40 (the first byte of the type-specific portion of the queue entry).

pbKey2
cbKey2

describe a second key field to be compared with a string located at an offset oKey2 in the queue entry.

oKey2

is the offset of the second string key field in the queue entry. The offset starts from byte 40 (the first byte of the type-specific portion of the queue entry).

pEntryRet
sEntryRet

describe the buffer into which the queue entry is read.

pStatusBlock
sStatusBlock

describe the buffer where the status block for the queue entry is returned.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	4
1	RtCode	1	0
2	nReqPbCb	1	3
3	nRespPbCb	1	2
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	142
12	oKey1	2	
14	oKey2	2	
16	pbQueueName	4	
20	cbQueueName	2	
22	pbKey1	4	
26	cbKey1	2	
28	pbKey2	4	
32	cbKey2	2	
34	pEntryRet	4	
38	sEntryRet	2	
40	pStatusBlock	4	
44	sStatusBlock	2	

This page intentionally left blank

MarkNextQueueEntry (pbQueueName, cbQueueName, fReturnIfNoEntries, pEntryRet, sEntryRet, pStatusBlock, sStatusBlock): ercType

Description

MarkNextQueueEntry is used by the queue server to read the first unmarked entry from the specified queue into a buffer for processing. Entries are marked in order of priority.

MarkNextQueueEntry marks the entry as being in use and returns a queue entry handle by which the entry is identified in a subsequent call to RemoveMarkedQueueEntry.

Procedural Interface

MarkNextQueueEntry (pbQueueName, cbQueueName, fReturnIfNoEntries, pEntryRet, sEntryRet, pStatusBlock, sStatusBlock): ercType

where

pbQueueName
cbQueueName

describe a queue name corresponding to the queue name specified when the queue was installed.

fReturnIfNoEntries

is a flag indicating the conditions under which the Queue Manager responds to MarkNextQueueEntry.

TRUE means the Queue Manager returns status code 903 ("No entries available") if no unmarked entry is queued.

FALSE means the Queue Manager responds only if an unmarked entry is queued. The Queue Manager must be informed that the queue server's partition is being vacated: the queue server can issue the TerminateQueueServer request before exiting, or a VacatePartition request can be issued for the queue server's partition.

pEntryRet
sEntryRet

describe the buffer into which the queue entry is read.

pStatusBlock
sStatusBlock

describe the buffer where the queue status block for the queue entry is returned.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntlnfo	1	2
1	RtCode	1	0
2	nReqPbCb	1	1
3	nRespPbCb	1	2
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	141
12	fReturnIfNoEntries	1	
13	reserved	1	
14	pbQueueName	4	
18	cbQueueName	2	
20	pEntryRet	4	
24	sEntryRet	2	
26	pStatusBlock	4	
30	sStatusBlock	2	

McrVersion (pVerStruct, sVerStruct, pcbRet): ercType

Description

McrVersion returns a structure that identifies the version of the installed Magnetic Card Reader (MCR) Service.

The MCR Service controls a magnetic card reader. This device reads the magnetic strip on a credit card, identification card, or other similar medium.

Procedural Interface

McrVersion (pVerStruct, sVerStruct, pcbRet): ercType

where

pVerStruct
sVerStruct

describe a memory area to which the structure containing the version is written. The format of the structure is as follows:

Offset	**Field**	**Size (Bytes)**
0	version	2

pcbRet

describes a word where the actual length of the returned structure is placed.

Request Block

scbRet is always 2.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	2
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	8228
12	reserved	6	
18	pVerStruct	4	
22	sVerStruct	2	
24	pcbRet	4	
28	scbRet	2	2

MediateIntHandler (fDeviceInt): ercType

Description

The MediateIntHandler primitive converts a raw interrupt handler to a mediated interrupt handler. Before using MediateIntHandler, the raw interrupt handler must save the contents of the registers of the interrupted process on the stack of the interrupted process in the following order: AX, BX, DS, CX, ES, SI, DX, DI, BP.

After that, the argument to the call should then be saved on the stack. No other information can be saved on the stack at the time of the call to MediateIntHandler. MediateIntHandler switches the stack (SS:SP) to point to the MIH stack.

Procedural Interface

MediateIntHandler (fDeviceInt): ercType

where

fDeviceInt

is TRUE or FALSE. TRUE (0FFh) indicates the interrupt handler serves device-generated interrupts. FALSE (0) indicates the interrupt handler serves software-generated interrupts (traps).

Request Block

MediateIntHandler is a Kernel primitive.

This page intentionally left blank

MenuEdit (pNlsTableArea, iFrame, iCol, iLine, pbPrompt, cbPrompt, pFieldDesc, pChRet): ercType

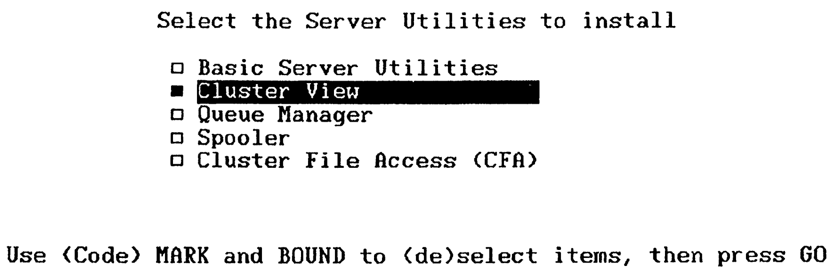
Description

MenuEdit displays a menu from the user can select one or more items, and processes the user responses. The menu can display up to 16 items. To select items, the user highlights them using the **Up Arrow**, **Down Arrow**, **Return**, and **Next** keys, or the **Mark** and **Bound** keys. To deselect an item, **Code-Bound** is used. When satisfied with the selection(s) made, the user presses **Go**. If the Mouse Service is installed, a mouse may be used to highlight the items.

Note that exit responses must be handled by the programmer using MenuEdit. For example, if the user pressed **Cancel** (not the correct exit response), the program might respond by beeping.

MenuEdit processes each field separately based on the field descriptor for that field. For each field, there may be an associated online help message or a user-written procedure that handles online help. (For an example of an online help message displayed when using a message file, see the description of the FormEdit operation. This operation displays message file help messages using the same format.)

When MenuEdit is called, a menu such as the one shown in Figure 3-3 is displayed.



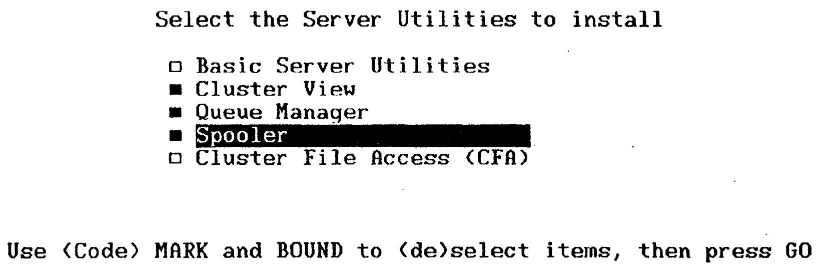
```
        Select the Server Utilities to install

         □ Basic Server Utilities
         ■ Cluster View
         □ Queue Manager
         □ Spooler
         □ Cluster File Access (CFA)


Use (Code) MARK and BOUND to (de)select items, then press GO
```

Figure 3-3. Menu With One Item Selected

In Figure 3-3, the first line is an (optional) message that prompts the user to select from the server utilities to install. To select a single utility, user can press **Right Arrow**, **Left Arrow**, or **Next** to highlight utilities, one at a time, until the desired one is highlighted.

Alternately, the user can select several utilities. Let's assume the user would like to select 'Cluster View', 'Queue Manager', and 'Spooler'. To do so, the user can first press **Mark** to highlight 'Cluster View'. When 'Cluster View' is highlighted, the square box to the left of this highlighted choice is filled, as shown in Figure 3-3.

To also select 'Queue Manager' and 'Spooler', the user moves through the menu by pressing **Down Arrow** or **Next** until 'Spooler' is reached. Then, by pressing **Bound**, all items ('Cluster View' through 'Spooler') are highlighted and the square box to the left of each highlighted choice is filled, as shown in Figure 3-4.



```
Select the Server Utilities to install

 □ Basic Server Utilities
 ■ Cluster View
 ■ Queue Manager
 ■ Spooler
 □ Cluster File Access (CFA)


Use (Code) MARK and BOUND to (de)select items, then press GO
```

Figure 3-4. Menu With Three Items Selected

This same procedure works in reverse. That is, 'Spooler' cound have been selected first and 'Cluster View', last.

To deselect individual items from a group already selected, the user can highlight the appropriate item and press **Code-Mark**. Doing so removes the highlight and the plus sign preceding the item. For example, the user may want to deselect Queue Manager from the group shown above.

MenuEdit process menu items based on default values or on tables in the Native Language Support (NLS) table area. See the description of *pNlsTableArea*. (For details on NLS, see "Native Language Support" in the *CTOS Operating System Concepts Manual*.)

The MenuEdit operation is typically used by a software installation program to obtain information from the user about how an installation should be customized. (For details on installation, see the *CTOS Programming Guide*.)

Procedural Interface

MenuEdit (pNlsTableArea, iFrame, iCol, iLine, pbPrompt, cbPrompt, pFieldDesc, pChRet): ercType

where

pNlsTableArea

MenuEdit returns a different value based on the following:

- If *pNlsTableArea* is 0 and [Sys]<Sys>Nls.sys was not present at system boot, MenuEdit processes the menu items based on default values.
- If *pNlsTableArea* is 0 and [Sys]<Sys>Nls.sys was present at system boot, MenuEdit processes menu items based on the NLS Character Class table.
- If *pNlsTableArea* is the memory address of an alternate set of NLS tables, MenuEdit processes menu items based on the NLS Character Class table in the alternate set of NLS tables.

iFrame

specifies the frame.

iCol

specifies the horizontal coordinate (word) within *iFrame* at which the first character of the menu is to be displayed.

iLine

specifies the vertical coordinate (word) within *iFrame* at which the first character of the menu is to be displayed.

pbPrompt
cbPrompt

describe an (optional) prompt displayed within the menu. The prompt can be used, for example, to provide the user with general instructions on how to fill out the form and to indicate which key the user must press to exit the menu. The menu shown in Figure 3-3 and Figure 3-4 displays the following prompt:

```
Select the Server Utilities to install
```

If *cbPrompt* > 0, the prompt is the first string displayed in the menu.

pFieldDesc

is the memory address of a menu field descriptor. The descriptor is either of two versions. One version is used with message files and the other, with message strings stored in memory.

The format of each of these descriptor types is described below:

Using Message Files

Field	**Size (Bytes)**	**Description**
bFieldType	1	is always the value 4.
sFieldDesc	2	is 35 (the size in bytes of this descriptor).
fMultiSelect	1	is a flag that is TRUE if the user selects more than one menu item or FALSE if only one item is selected.
prgimsgChoices	4	is the memory address of an array of 1-byte message numbers for the field name strings, such as Cluster View', 'Spooler,' and so forth in the examples.
nChoices	2	is the number of choices in the menu, such as 4 for the example menu.
sItemMax	2	is the maximum number of bytes in a field name. This value is used to determine the width of the area to highlight when the user selects a choice.

Using Message Files *(continued)*

Field	Size (Bytes)	Description
pChoiceRet	4	is the memory address of a word of boolean bits indicating the choice(s) selected by the user. If an item is selected, the corresponding bit is set. Bit 0 is set if the first item (that is, the topmost item in the list) is selected, bit 1 is set if the second item is selected, and so forth. It is the responsibility of the programmer to mask bits to determine which bit(s) are set.
sChoiceRet	2	is the size in bytes of the choice(s) selected. If *sChoiceRet* is 1, one choice was selected.
pWorkArea	4	is the memory address of an internal work area used by MenuEdit for this field.
sWorkArea	2	is the work area size in bytes. The minimum size is 150 bytes plus the total sum of the bytes in the choice strings.
pHelpProc	4	is the memory address of a user-written help procedure called by MenuEdit or is 0. If *pHelpProc* is not 0, *pHelpProc* is the address of a user procedure of the following format: Call Proc(*wHelpToken*, *pErcRet*)

Using Message Files *(continued)*

Field	Size (Bytes)	Description
		wHelpToken is described in the next field of this descriptor. (See *wHelpToken* below.) *pErcRet* is the address of a status code returned indirectly to FormEdit.
		If, however, *pHelpProc* is 0, a help message is displayed to the screen from a help message file. The fields for the help messages are *iMsgHelp*, *iMsgHelpPrompt*, and *iMsgHelpTitle*. (These fields are described below.)
wHelpToken	2	is a number passed to the user-written help procedure for use in displaying a custom help menu.
iMsgHelp	2	is the message number of the help message to be displayed in the middle of the help box. (See the description of the FormEdit operation. It describes the help message.)
iMsgHelpPrompt	2	is message number of the prompt message to be displayed below the help message. (See the FormEdit operation for details.)
iMsgHelpTitle	2	is message number of a help title to be displayed on the top border of the help box. (See the FormEdit operation for details.)

Using Message Strings Stored In Memory

Field	Size (Bytes)	Description
bFieldType	1	is always 204.
sFieldDesc	2	is 35 (the size in bytes of this descriptor).
prgSdChoices	4	is the memory address of an array of Sd-type (6-byte) records describing the choices that can be selected. For each record, the first 4 bytes contain the memory address of the choice string, and the last 2 contain the string byte count. If, for example, 'Spooler' is one of the choices, *prgSdChoices* would contain an Sd-type record of the following form for this choice: a 4-byte pointer to the string 'Spooler' a 2-byte word containing the byte count value of 7
nChoices	2	is the number of choices in the menu. The menu in Figure 3-3 and Figure 3-4 shows 5 choices.
sItemMax	2	is the maximum number of bytes in a field name. This value is used to determine the width of the area to highlight when the user selects a choice.

Using Message Strings Stored In Memory *(continued)*

Field	Size (Bytes)	Description
pChoiceRet	4	is the memory address of an array of bytes the values of which indicate the choice(s) selected by the user. Numbering is consecutive starting with 0. The first choice is 0, the second is 1, and so forth. For instance, if the user selected 'Cluster View' in the example menu, 1 would be returned; If 'Cluster View' and 'Queue Manager' were selected, the array values would be 1 (second array element) and 3 (third element).
sChoiceRet	2	is the size in bytes of the choice(s) selected. If *sChoiceRet* is 1, one choice was selected.
pWorkArea	4	is the memory address of an internal work area used by MenuEdit for this field.
sWorkArea	2	is the work area size in bytes. The minimum size is 150 bytes plus the sum of the bytes in the choice strings.

Using Message Strings Stored In Memory *(continued)*

Field	Size (Bytes)	Description
pHelpProc	4	This field and those that follow are the same five help fields as described in the previous descriptor. See "Using Message Files."
wHelpToken	2	See *pHelpProc*.
iMsgHelp	2	See *pHelpProc*.
iMsgHelpPrompt	2	See *pHelpProc*.
iMsgHelpTitle	2	See *pHelpProc*.

pChRet

is the memory address of the byte where the exit character is to be written. The exit character is used to determine if the menu should be exited (such as when the user presses **Go** in the example menu) or if some other appropriate action should be taken.

Request Block

MenuEdit is an object module procedure.

Mode3DmaReload: ercType

Caution: *This operation works on workstation hardware only. Do not use it on shared resource processor hardware.*

Description

Mode3DmaReload is used to set up and program Direct Memory Access (DMA) to and from X-Bus memory master devices, such as the HB-001, TM-001, VP-002, and XE-002, using X-Bus mode 3 DMA.

The first call to Mode3DmaReload initializes the 8237 DMA chip on NGENs running on prior operating system versions. This initialization is done in the operating system in later versions.

Mode3DmaReload is called by system services for X-Bus memory master devices often enough so that 65,535 bytes of data are not transferred between calls. The operation reprograms the countdown register of the 8237 DMA chip to avoid terminal count (which would cause a parity error, bus timeout, or other erroneous memory read or write) on NGENs running on prior operating system versions. This reprogramming is done in the operating system in later versions.

Status code 121 ("Old OS") is returned when running on operating system versions prior to 9.1.

Procedural Interface

Mode3DmaReload: ercType

Request Block

Mode3DmaReload is an object module procedure.

This page intentionally left blank

MountVolume (pbDevSpec, cbDevSpec, pbDevPassword, cbDevPassword): ercType

Description

MountVolume mounts the volume on the specified disk drive.

Mounting (and dismounting) of volumes is normally controlled by the Automatic Volume Recognition (AVR) capability of the file management system. The Mount (and Dismount) operations are provided for the use of commands, such as **Format Disk**, that must override AVR. (**Format Disk** is described in the *CTOS Executive Reference Manual.*)

Procedural Interface

MountVolume (pbDevSpec, cbDevSpec, pbDevPassword, cbDevPassword): ercType

where

pbDevSpec
cbDevSpec

describe a character string of the form {Node}[DevName]. Square brackets are optional for the device name. The distinction between uppercase and lowercase is not significant in matching device names.

pbDevPassword
cbDevPassword

describe parameter values that are ignored. Use zeros for these parameters.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	2
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	11
12	reserved	6	
18	pbDevSpec	4	
22	cbDevSpec	2	
24	pbDevPassword	4	
28	cbDevPassword	2	

MouseVersion (pVerStruct, sVerStruct, pcbRet): ercType

Description

MouseVersion returns a structure that identifies the version of the installed Mouse Service.

Procedural Interface

MouseVersion (pVerStruct, sVerStruct, pcbRet): ercType

where

pVerStruct
sVerStruct

describe a memory area to which the structure containing the version is written. The format of the structure is as follows:

Offset	**Field**	**Size (Bytes)**
0	version	2

pcbRet

describes a word into which the actual length of the returned structure is placed.

Request Block

scbRet is always 2.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	2
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	8212
12	reserved	6	
18	pVerStruct	4	
22	sVerStruct	2	
24	pcbRet	4	
28	scbRet	2	2

MoveFrameRectangle (iSrcFrame, iDstFrame, iSrcCol, iSrcLine, iDstCol, iDstLine, cCols, cLines, fBlankSrc): ercType

NOTE: This operation is supported by protected mode operating systems only.

Description

MoveFrameRectangle moves an arbitrary rectangle of characters and corresponding attributes within a frame of the character map to another position in the map. This operation is used, for example, to move a block of text from one area on a screen to another.

Procedural Interface

MoveFrameRectangle (iSrcFrame, iDstFrame, iSrcCol, iSrcLine, iDstCol, iDstLine, cCols, cLines, fBlankSrc): ercType

where

iSrcFrame

specifies the frame from which the characters/attributes will be moved. If *iSrcFrame* = 255, the coordinates *iSrcCol* and *iSrcLine* refer to virtual screen coordinates. This works exactly the same as if the caller has a frame that overlays the real screen.

iDstFrame

specifies the frame where the characters/attributes will be moved. If *iDstFrame* = TRUE, the coordinates *iDstCol* and *iDstLine* refer to virtual screen coordinates.

iSrcCol
iSrcLine

specify the horizontal and vertical coordinates of the top left corner of the rectangle within *iSrcFrame*.

iDstCol
iDstLine

specify the horizontal and vertical coordinates of the top left corner of the rectangle destination within *iDstFrame*.

cCols

is the number of columns in the rectangle to be moved.

cLines

is the number of lines in the rectangle to be moved.

fBlankSrc

should be TRUE if the source field is to be blanked after the move. The character used to fill with blanks is taken from the Video Control Block (VCB) (vcb.vSpace).

NOTE: If the position of the destination rectangle is such that part of the rectangle lies outside the destination frame, status code 500 ("Coordinates do not agree") is returned and the characters/attributes are clipped on the screen.

Request Block

MoveFrameRectangle is a system-common procedure.

MoveOverlays (pOvZoneNew): ercType

Description

MoveOverlays changes the location of the overlay zone in a program that is using the Virtual Code facility.

You should use this operation, for example, if your program has other entities such as heaps that are competing for memory space. In such a case, you may need to shift the overlay zone.

Procedural Interface

MoveOverlays (pOvZoneNew): ercType

where

pOvZoneNew

is the memory address to which the overlay zone should be moved. *pOvZoneNew* must be paragraph aligned.

Request Block

MoveOverlays is an object module procedure.

This page intentionally left blank

NameAllocClass (pClassIdRet, bFlags): ercType

NOTE: This operation only works on virtual memory operating systems.

Description

NameAllocClass returns a unique 16-bit class identifier (ID). A class is a name table within which names must be unique.

For details on name management, see "Managing Names" in the section entitled "Utility Operations" in the *CTOS Operating System Concepts Manual.*

Procedural Interface

NameAllocClass (pClassIdRet, bFlags): ercType

where

pClassIdRet

is the memory address to which the unique class ID (word) is written.

bFlags

is a byte. The bits are interpreted as follows:

Bit	**Meaning**
0	Set for case sensitivity within the class. Clear for case insensitivity.
1-7	Reserved.

Request Block

sClassIdRet is always 2.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	8
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	476
12	bflags	1	
13	reserved	7	
20	pClassIdRet	4	
24	sClassIdRet	2	2

NameQuery (pbName, cbName, wClass, tyName, pTagRet): ercType

NOTE: This operation only works on virtual memory operating systems.

Description

NameQuery returns the 32-bit tag associated with the specified name in the specified class.

For details on name management, see "Managing Names" in the section entitled "Utility Operations" in the *CTOS Operating System Concepts Manual*.

Procedural Interface

NameQuery (pbName, cbName, wClass, tyName, pTagRet): ercType

where

pbName
cbName

describe the name string.

wClass

is the 16-bit class identifier.

tyName

is reserved for future use. This value should be 0.

pTagRet

is the memory address to which the-32 bit tag is written.

Request Block

sTagRet is always 4.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	8
1	RtCode	1	0
2	nReqPbCb	1	1
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	411
12	wClass	2	
14	tyName	1	
15	reserved	5	
20	pbName	4	
24	cbName	2	
26	pTagRet	4	
30	sTagRet	2	4

NameRegister (pbName, cbName, wClass, tyName, tag): ercType

NOTE: This operation only works on virtual memory operating systems.

Description

NameRegister associates a 32-bit tag with the specified name. The operation adds the name and tag to the specified class in the name heap.

For details on name management, see "Managing Names" in the section entitled "Utility Operations" in the *CTOS Operating System Concepts Manual.*

Procedural Interface

NameRegister (pbName, cbName, wClass, tyName, tag): ercType

where

pbName
cbName

describe the name.

wClass

is the 16-bit class ID returned by a previous call to the NameAllocClass operation. (See NameAllocClass.)

tyName

is reserved for future use. This value should be 0.

tag

is the 32-bit tag value associated with the name.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	8
1	RtCode	1	0
2	nReqPbCb	1	1
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	410
12	wClass	2	
14	tyName	1	
15	reserved	1	
16	tag	4	
20	pbName	4	
24	cbName	2	

NameRemove (pbName, cbName, wClassId, tyName): ercType

NOTE: This operation only works on virtual memory operating systems.

Description

NameRemove removes the specified name from the specified class in the name heap.

For details on name management, see "Managing Names" in the section entitled "Utility Operations" in the *CTOS Operating System Concepts Manual.*

Procedural Interface

NameRemove (pbName, cbName, wClassId, tyName): ercType

where

pbName
cbName

describe the name to be removed.

wClassId

is the 16-bit ID of the class containing the name.

tyName

is a byte reserved for future use. This parameter should be 0.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	8
1	RtCode	1	0
2	nReqPbCb	1	1
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	412
12	wClassId	2	
14	tyName	1	
15	reserved	5	
20	pbName	4	
24	cbName	2	

NewProcess (pProcessStruct, userNum, pPidRet): ercType

Description

The NewProcess primitive is similar to CreateProcess in that it creates a new process and schedules it for execution.

NewProcess differs from CreateProcess in that a user number and the initial values of all the registers may be specified, and a process ID is returned. The process ID may be used as a parameter to other process management operations.

The priority specified by NewProcess may be modified by SetDeltaPriority before the process is scheduled for execution. (See SetDeltaPriority.)

Procedural Interface

NewProcess (pProcessStruct, userNum, pPidRet): ercType

where

pProcessStruct

is the memory address of the following structure:

Offset	Field	Size (Bytes)	Description
0	pEntry	4	Memory address (CS:IP) at which to begin execution of the new process. This address is usually an operation address.
4	saData	2	Value of the segment base address to be used in accessing the new process's global data. This value is loaded into the Data Segment (DS) register when the process is scheduled for execution.*
6	saExtra	2	Value to be loaded into the Extra Segment (ES) register when the new process is scheduled for execution. This value must be a valid segment base address or 0.
8	saStack	2	Value of the segment base address of the new process's stack. This value is loaded into the Stack Segment (SS) register when the process is scheduled for execution.*

*DS and Stack Segment (SS) register values must be the same in the medium model of computation, which is used for all operating system programs. If the newly created process intends to share data with the calling process, the DS and SS values of the newly created and the calling processes must be the same: the current value in the calling process's DS register must be loaded into the new process's DS *and* SS registers.

Offset	Field	Size (Bytes)	Description
10	oStackInit	2	Value of the offset of the last word in a global array that is declared in the caller's data space. This value is loaded into the Stack Pointer (SP) register when the new process is scheduled for execution. The array provides memory for the newly created process to use as a stack.
12	priority	1	Priority at which the new process is to be scheduled for execution. The highest priority is 0; the lowest is 254.
13	fSys	1	Always FALSE.
14	defaultRespExch	2	To make operating system calls from the new process, allocate an exchange using the AllocExch operation and provide it here. The exchange becomes the default response exchange of the new process. To avoid possible conflict, the calling process must never use this exchange again.
16	fDebug	1	If TRUE, the Debugger is invoked before executing the new process's first instruction. If FALSE, the process is scheduled for execution. (See the *CTOS Debugger User's Guide.*)
18	AX	2	Initial value of AX register
20	BX	2	Initial value of BX register
22	CX	2	Initial value of CX register
24	DX	2	Initial value of DX register
26	SI	2	Initial value of SI register
28	DI	2	Initial value of DI register

userNum

is 0 (the user number of the calling program is used).

pPidRet

is the memory address of a word to which the process ID of the newly created process is returned.

Request Block

NewProcess is a Kernel primitive.

NlsCase (pNlsTableArea, bCharIn, pbCharOut, fUpper): ercType

NOTE: To ensure portability across different language environments, an application should use EnlsCase rather than NlsCase. (See EnlsCase.)

Description

NlsCase translates a given character from lowercase to uppercase, or from uppercase to lowercase.

Procedural Interface

NlsCase (pNlsTableArea, bCharIn, pbCharOut, fUpper): ercType

where

pNlsTableArea

NlsCase returns a different value based on the following:

- If *pNlsTableArea* is 0 and [Sys]<Sys>Nls.sys was not present at system boot, an answer is returned based on the American character set.
- If *pNlsTableArea* is 0 and [Sys]<Sys>Nls.sys was present at system boot, an answer is returned based on the NLS Lowercase to Uppercase or Uppercase to Lowercase table.
- If *pNlsTableArea* is the address of an alternate set of NLS tables, an answer is returned based on the NLS Lowercase to Uppercase or Uppercase to Lowercase table in the alternate set of NLS tables.

bCharIn

is the character to be translated.

pbCharOut

is the address of a byte to which the returned translated character is placed.

fUpper

is either TRUE, meaning lowercase to uppercase translation, or FALSE, meaning uppercase to lowercase translation.

Request Block

NlsCase is an object module procedure.

NlsClass (pNlsTableArea, bCharIn, pbClassRet): ercType

NOTE: To ensure portability across different language environments, an application should use EnlsClass rather than NlsClass. (See EnlsClass.)

Description

NlsClass takes a given character and returns the class of that character.

Procedural Interface

NlsClass (pNlsTableArea, bCharIn, pbClassRet): ercType

where

pNlsTableArea

NlsClass returns a different value based on the following:

- If *pNlsTableArea* is 0 and [Sys]<Sys>Nls.sys was not present at system boot, an answer is returned based on the American character set.
- If *pNlsTableArea* is 0 and [Sys]<Sys>Nls.sys was present at system boot, an answer is returned based on the NLS Character Class table.
- If *pNlsTableArea* is the address of an alternate set of NLS tables, an answer is returned based on the NLS Class Character table in the alternate set of NLS tables.

bCharIn

is the character to be classified.

pbClassRet

is the address of a byte to which the class is returned. The value returned will be one of the following:

Value	**Meaning**
0	Numeric
1	Alpha
2	Special (Non-Alphanumeric but commonly displayed)
3	Graph (Line drawing and other special graphics)
4	Blind (Not generally intended for display)

Request Block

NlsClass is an object module procedure.

NlsCollate (pNlsTableArea, pbString1, cbString1, pbString2, cbString2, pbCollResRet, pbPrioResRet): ercType

Description

NlsCollate compares two strings to determine if they are equal or if one is greater than the other.

Unlike the NlsULCmpB operation, which compares the string characters based on their ASCII values, this operation allows you to compare characters based on other criteria as well, as described below. The ULCmpB operation, which also compares two strings, is documented for historical reasons only and should not be used.

NlsCollate uses the NLS collation structure. This structure consists of one mandatory character collation table, an optional character priority table, and tables for the definitions of 1-for-2 and 2-for-1 substitutions. If no collation structure is present, ordering is based on ASCII character codes, except that lowercase roman letters (a...z) are first converted to their uppercase equivalents (A...Z).

The procedural steps to determining the resulting collation and priority ordering of two strings are, as follows:

- Character pairs represented by a single character code are identified and replaced by their singular code (1-for-2 substitution). For example, the Spanish "ch" can be substituted by a single code causing it to be collated after "c" but before "d."

- Single characters represented by two codes are identified and replaced by their two codes (2-for-1 substitution). For example, the German "sharp-s," can be collated as "ss" and a-umlaut as "au."

- All remaining characters in the strings are identified and replaced by their character codes found in the collation or priority table (simple substitution). This allows for redefinition of the sort order.

- The strings are collated according to the character collation table. If they collate as equal, string priority is determined by the character priority table.

  This is sometimes used for case differences and, more commonly, for accent mark differences. For example, the vowel "e" is considered equal in all its forms except for priority, which alternates between uppercase and lowercase versions, first with no accents, then with acute, grave, circumflex, and umlaut.

  As a simple example of this step, the following strings

  aaaezEz

  aaaEzez

  collate as equal. However, using the collation structure, "E" may be defined as having a higher priority than "e." When the character priority table is applied, the second string above collates before the first.

Procedural Interface

NlsCollate (pNlsTableArea, pbString1, cbString1, pbString2, cbString2, pbCollResRet, pbPrioResRet): ercType

where

pNlsTableArea

NlsCollate returns a different value based on the following:

- If *pNlsTableArea* is 0 and [Sys]<Sys>Nls.sys was not present at system boot, an answer is returned based on the ordering of ASCII character codes, except that lowercase roman letters (a...z) are first converted to their uppercase equivalents (A...Z).

- If *pNlsTableArea* is 0 and [Sys]<Sys>Nls.sys was present at system boot, an answer is returned based on the NLS Collation tables.
- If *pNlsTableArea* is the memory address of an alternate set of NLS tables, an answer is returned based on the NLS Collation tables in the alternate set of NLS tables.

pbString1
cbString1

describe the first string used. *String1* is not modified by NlsCollate.

pbString2
cbString2

describe the second string used. *String2* is not modified by NlsCollate.

pbCollResRet

is the memory address of a 1-byte area to which a code indicating the result of the string collation is placed. The code values are described as follows:

Code	**Description**
0	string1 < string2
1	string1 = string2
2	string1 > string2

pbPrioResRet

is the memory address of a 1-byte area to which a code indicating the result of collation based upon character priority is placed. If *pbCollResRet* equals 1 (string equality), *pbPrioResRet* determines string priority. Otherwise, *pbPrioResRet* equals *pbCollResRet*. The *pbPrioResRet* code values are the same as those shown above for *pbCollResRet*.

Request Block

NlsCollate is an object module procedure.

NlsFormatDateTime (pNlsTableArea, dateTime, pbTemplate, cbTemplate, pbRet, cbMax, pcbRet): ercType

Description

NlsFormatDateTime converts the date and time from the System Date/Time format (described in "Utility Operations" in the *CTOS Operating System Concepts Manual*) to textual (string) format. The conversion is controlled by control letters embedded in a user-supplied date and time format template.

Programs should use NlsFormatDateTime or NlsStdFormatDateTime instead of FormatDateTime. FormatDateTime is documented for historic reasons only. The difference between NlsFormatDateTime and NlsStdFormatDateTime is that you supply the template to the former; the latter accepts a template from the NLS Date and Time Formats table and, therefore, is more easily nationalized. (For details, see "Native Language Support" in the *CTOS Operating System Concepts Manual.*)

The date and time format template is user supplied. If too few template control letters are defined in the NLS date and time formats table, status code 13507 ("Bad date table") is returned. This can occur, for example,

- if a user inadvertently deletes a character when redefining the control letters in Nls.sys
- if a new version of NlsFormatDateTime, requiring newly defined control letters is used with an old set of NLS tables

User-supplied templates are interpreted in the same manner as the format templates in Appendix E. (See Appendix E for a summary of rules on how NlsFormatDateTime interprets the format templates.)

If the template described by *pbTemplate* and *cbTemplate* is improperly constructed, the violating portion of the template is put into the returned date and time string with no expansion or substitution. If, for example, the template provided includes !*tn!, FormatDateTime would move this part of the template verbatim to the output string, since the control letter is established as the letter t, and an additional control letter n is included in the template.

If the template is properly constructed but contains a date escape sequence with an undefined control letter, that escape sequence expands to the error command form !x?! (where x is the undefined letter) in the output string. If, for example, the control letter b is undefined, !B*b! results in !b?!.

Procedural Interface

NlsFormatDateTime (pNlsTableArea, dateTime, pbTemplate, cbTemplate, pbRet, cbMax, pcbRet): ercType

where

pNlsTableArea

NlsFormatDateTime returns a different value based on the following:

- If pNlsTableArea is 0 and [Sys]<Sys>Nls.sys was not present at system boot, American date and time names are returned.
- If pNlsTableArea is 0 and [Sys]<Sys>Nls.sys was present at system boot, an answer is returned based on the NLS Date and Time Formats table.
- If pNlsTableArea is the memory address of an alternate set of NLS tables, an answer is returned based on the NLS Date and Time Formats table in the alternate set of NLS tables.

dateTime

is the 32-bit date/time in the System Date/Time format.

pbTemplate
cbTemplate

describe the memory area containing the template to be used to control formatting.

pbRet
cbMax

describe the memory area to which the formatted date and time string is returned. If *cbMax* is not large enough to contain the resultant string, the string is truncated.

pcbRet

is the memory address of the word to which the number of bytes returned is placed.

Request Block

NlsFormatDateTime is an object module procedure.

This page intentionally left blank.

NlsGetYesNoStrings (pNlsTableArea, prgSdStringsRet): ercType

Description

NlsGetYesNoStrings returns the strings defined in the Native Language Support (NLS) Yes or No Strings table. (See the description of *pNlsTableArea*. For details on the NLS tables, see "Native Language Support" in the *CTOS Operating System Concepts Manual*.)

Procedural Interface

NlsGetYesNoStrings (pNlsTableArea, prgSdStringsRet): ercType

where

pNlsTableArea

NlsGetYesNoStrings returns a different value based on the following:

- If *pNlsTableArea* is 0 and [Sys]<Sys>Nls.sys was not present at system boot, NlsGetYesNoStrings returns an answer based on the default strings Yes and No.
- If *pNlsTableArea* is 0 and [Sys]<Sys>Nls.sys was present at system boot, NlsGetYesNoStrings returns an answer based on the NLS Yes or No Strings table.
- If *pNlsTableArea* is the memory address of an alternate set of NLS tables, NlsGetYesNoString returns an answer based on the NLS Yes or No Strings table in the alternate set of NLS tables.

prgSdStringsRet

describes a buffer to be filled with two sdType values. The first sdType value describes the Yes string. The second describes the No string. Each sdType value is a 6 byte quantity in which the first 4 bytes contain the address of the string, and the last 2 bytes contain the string length.

Request Block

NlsGetYesNoStrings is an object module procedure.

NlsGetYesNoStringSize (pNlsTables, prgcbStringsRet): ercType

Description

NlsGetYesNoStringSize returns the sizes of the strings defined in the Native Language Support (NLS) Yes and No Strings table. (See the description of *pNlsTableArea*. For details on the NLS tables, see "Native Language Support" in the *CTOS Operating System Concepts Manual*.)

Procedural Interface

NlsGetYesNoStringSize (pNlsTableArea, prgcbStringsRet): ercType

where

pNlsTableArea

NlsGetYesNoStringSize returns a different value based on the following:

- If *pNlsTableArea* is 0 and [Sys]<Sys>Nls.sys was not present at system boot, NlsGetYesNoStringSize returns an answer based on the default strings Yes and No.
- If *pNlsTableArea* is 0 and [Sys]<Sys>Nls.sys was present at system boot, NlsGetYesNoStringSize returns an answer based on the NLS Yes or No Strings table.
- If *pNlsTableArea* is the memory address of an alternate set of NLS tables, NlsGetYesNoStringSize returns an answer based on the NLS Yes or No Strings table in the alternate set of NLS tables.

prgcbStringsRet

describes a buffer to be filled with two word values. The first element describes the length of the Yes string. The second element describes the length of the No string.

Request Block

NlsGetYesNoStringSize is an object module procedure.

NlsNumberAndCurrency (pNlsTableArea, ppTableRet): ercType

Description

NlsNumberAndCurrency returns the address of the Number and Currency Formats table.

Procedural Interface

NlsNumberAndCurrency (pNlsTableArea, ppTableRet): ercType

where

pNlsTableArea

NlsNumberAndCurrency returns a different value based on the following:

- If *pNlsTableArea* is 0 and [Sys]<Sys>Nls.sys was not present at system boot, an answer is returned based on the American number and currency symbols.
- If *pNlsTableArea* is 0 and [Sys]<Sys>Nls.sys was present at system boot, an answer is returned based on the NLS Number and Currency Formats table.
- If *pNlsTableArea* is the memory address of an alternate set of NLS tables, an answer is returned based on the NLS Number and Currency Formats table in the alternate set of NLS tables.

ppTableRet

is the address of a four byte memory area to which the address of the number and currency table will be placed. The number and currency table has the following format:

Offset	**Meaning**	**Size (Bytes)**
0	Decimal character	1
1	Triad character	1
2	First triad flag	1
3	List separator character	1
4	Currency position	1
5	Currency symbol (sbString)	1-4

Request Block

NlsNumberAndCurrency is an object module procedure.

NlsParseTime (pNlsTableArea, psbDateTime, pExpDateTimeStruct): ercType

Description

NlsParseTime converts a caller-supplied string into the Expanded Date/Time format. (For details on the expanded format, see "Expanded Date/Time Format" in Chapter 4, "System Structures.")

The operation fills in only those fields in the Expanded Date/Time format that are found in the string, with the following exceptions:

- If any field in the Expanded Date/Time format is initialized to have a value of -1 (0FFh for 1-byte fields or 0FFFFh for 2-byte fields), an error is returned if there is no entry in the string for that field. If there is more than one entry in the string for a field, the last one is used.

- If the field *dayOfMonth* in the Expanded Date/Time format is initialized to have a value of 0, status code 3806 ("Day of month is out of range 0 to 31") is returned.

Procedural Interface

NlsParseTime (pNlsTableArea, psbDateTime, pExpDateTimeStruct): ercType

where

pNlsTableArea

NlsParseTime returns a different value based on the following:

- If *pNlsTableArea* is 0 and [Sys]<Sys>Nls.sys was not present at system boot, an answer is returned based on the American date name strings.

- If *pNlsTableArea* is 0 and [Sys]<Sys>Nls.sys was present at system boot, an answer is returned based on the NLS Date Name table.

- If *pNlsTableArea* is the memory address of an alternate set of NLS tables, an answer is returned based on the NLS Date Name table in the alternate set of NLS tables.

psbDateTime

is the memory address of a string containing the date/time. The first byte of the string is the size of the string; subsequent bytes contain the characters in the string.

pExpDateTimeStruct

is the memory address of the Expanded Date/Time format, as returned by the ExpandDateTime operation, and/or initialized to have -1 as the value of each field.

Request Block

NlsParseTime is an object module procedure.

*NlsSpecialCharacters (pNlsTableArea, fOEMCharacters,
ppSpecialCharactersRet): ercType*

Description

NlsSpecialCharacters returns the address of the Native Language Support (NLS) Special Characters table. (For details on NLS, see the *CTOS Operating System Concepts Manual.*)

This operation is used by programs that require interpretation of special characters. An example of such a character is the literal insert character. Literal insert is defined in the table as 0A7h, and is usually generated by CODE-'. This character, as well as others in the Special Characters table, may be modified by editing the file Nls.asm.

The Special Characters table contains two regions: one containing special character entries to be used interally and a second, containing entries to be used by customers. The caller is returned the starting address of either of these regions.

Procedural Interface

*NlsSpecialCharacters (pNlsTableArea, fOEMCharacters,
ppSpecialCharactersRet): ercType*

where

pNlsTableArea

NlsSpecialCharacters returns a different value based on the following:

If *pNlsTableArea* is 0 and [Sys]<Sys>Nls.sys was not present at system boot, the pointer returned is the memory address of a default table based on American special characters. The default table contains no customer entries.

If *pNlsTableArea* is 0 and [Sys]<Sys>Nls.sys was present at system boot, the pointer returned is the memory address of the Special Characters table.

If *pNlsTableArea* is the memory address of an alternate set of Nls tables, the pointer returned is the memory address of the Special Characters table in the alternate set of Nls tables.

fOEMCharacters

is a flag that indicates whether the memory address returned is the starting address of the region for internal use or for customer use. TRUE means the starting address of the region for customer use is returned.

ppSpecialCharactersRet

is the memory address of a 4 byte memory area to which the address of the Special Characters table will be placed. The table has the following format:

Offset	**Field**	**Size (bytes)**	**Description**
0	nSysEntries	1	Number of entries for internal use
1..n*	rgbSysEntries	n	Array of internal use entries
n+1	nUserEntries	1	Number of entries for customer use
n+2	rgbUserEntries	m*	Array of internal use entries

**n=* *nSysEntries*
m= *nUserEntries*

Request Block

NlsSpecialCharacters is an object module procedure.

This page intentionally left blank.

NlsStdFormatDateTime (pNlsTableArea, iTemplate, datetime, pbRet, cbMax, pcbRet): ercType

Description

NlsStdFormatDateTime converts the date and time from the System Date/Time format (described in "Utility Operations" in the *CTOS Operating System Concepts Manual*) to textual (string) format. The conversion is controlled by control letters embedded in a template selected from the NLS Date and Time Formats table.

The Std phrase included in the name of this operation indicates that NlsStdFormatDateTime is considered the standard or preferred method for using date and time formatting. Using this operation provides for ease in nationalization. (For details, see "Native Language Support" in the *CTOS Operating System Concepts Manual.*) NlsStdFormatDateTime is similar to NlsFormatDateTime except that the latter accepts a user-supplied template and, thus, is less easily nationalized.

NlsStdFormatDateTime uses the templates as defined in Appendix E. If the Native Language Support (NLS) table area does not contain a template as specified by *iTemplate*, this operation informs the user by returning the message, "Invalid template index," instead of a date time string in the user's buffer at pbRet. No status code is returned.

Procedural Interface

NlsStdFormatDateTime (pNlsTableArea, iTemplate, datetime, pbRet, cbMax, pcbRet): ercType

where

pNlsTableArea

NlsStdFormatDateTime returns a different value based on the following:

- If *pNlsTableArea* is 0 and [Sys]<Sys>Nls.sys was not present at system boot, American date and time names are returned.
- If *pNlsTableArea* is 0 and [Sys]<Sys>Nls.sys was present at system boot, an answer is returned based on the NLS Date and Time Formats table.
- If *pNlsTableArea* is the memory address of an alternate set of NLS tables, an answer is returned based on the NLS Date and Time Formats table in the alternate set of NLS tables.

iTemplate

is either the index of the desired date and time format or the special value 0FFFFh. The value 0FFFFh means use the template designated within *[Sys]<Sys>Nls.sys* as the output template. If *[Sys]<Sys>Nls.sys* is not present, template number 5 is used. (The date and time templates are in Appendix E.)

datetime

is the 32-bit date and time in the System Date/Time format.

pbRet
cbMax

describe the memory area to which the formatted date and time string is returned. If *cbMax* is not large enough to accomodate the returned date and time string, the string is truncated.

pcbRet

is the memory address of the word to which the number of bytes is returned.

Request Block

NlsStdFormatDateTime is an object module procedure.

This page intentionally left blank

NlsULCmpB (pNlsTableArea, prgbString1, prgbString2, cbString, piNonCmpChrRet): ercType

Description

NlsULCmpB uses a case insensitive comparison to compare two strings for equality.

NlsULCmpB is identical to ULCmpB except that the former accepts the memory address of an NLS Uppercase to Lowercase table and returns a status code. NlsULCmpB should be used instead of ULCmpB for ease in nationalization. (For details, see "Native Language Support" in the *CTOS Operating System Concepts Manual.*)

Procedural Interface

NlsULCmpB (pNlsTableArea, prgbString1, prgbString2, cbString, piNonCmpChrRet): ercType

where

pNlsTableArea

NlsULCmpB returns a different value based on the following:

- If *pNlsTableArea* is 0 and [Sys]<Sys>Nls.sys was not present at system boot, an answer is returned based on the ordering of ASCII character codes, except that lowercase roman letters (a...z) are first converted to their uppercase equivalents (A...Z).
- If *pNlsTableArea* is 0 and [Sys]<Sys>Nls.sys was present at system boot, an answer is returned based on the NLS Uppercase to Lowercase table.
- If *pNlsTableArea* is the memory address of an alternate set of NLS tables, an answer is returned based on the NLS Uppercase to Lowercase table in the alternate set of NLS tables.

prgbString1
prgbString2

are memory addresses of two equal-length strings.

cbString

is the length of the two strings.

piNonCmpChrRet

is the memory address of a word to which the index of the first two characters in the strings that did not compare as equal is returned. If the strings are equal, 0FFFFh is returned.

Request Block

NlsULCmpB is an object module procedure.

NlsVerifySignatures (pNlsTableArea, cTables, prgwTableIds, prgwSignatures): ercType

Description

NlsVerifySignatures validates an alternate set of NLS tables, that is, it ensures that the signatures embedded within the provided alternate table area match those expected to be there.

The standard NLS tables provided in the file *[Sys]<Sys>Nls.sys* are validated by the operating system at system boot. If any of those tables are not valid, the operating system will make an entry in the PLog.

NlsVerifySignatures returns status code 13501 ("Bad Signature") if any of the tables signatures does not match the expected value.

Procedural Interface

NlsVerifySignatures (pNlsTableArea, cTables, prgwTableIds, prgwSignatures): ercType

where

pNlsTableArea

is either 0 (meaning use the NLS tables loaded at boot time), or is the address of an alternative set of NLS tables.

cTables

is the number of tables provided in the alternate table area.

prgwTableIds

is the memory address of the first byte of an array of NLS table ID codes.

prgwSignatures

is the memory address of the first byte of an array of words containing the expected NLS signatures.

Request Block

NlsVerifySignatures is an object module procedure.

NlsYesNoOrBlank (pNlsTableArea, pSdParam, pbCase): ercType

Description

NlsYesNoOrBlank performs a case insensitive comparison against nationalized words meaning yes or no.

NlsYesNoOrBlank is similar to NlsYesOrNo, except that, in addition, NlsYesNoOrBlank checks for a null string.

The string comparison will complete successfully if the string passed matches any portion of a yes or no word. For example, in an American system, "y," "ye," and "yes" will all match "yes."

It is recommended that this operation (or the related NlsYesOrNo) be used in conjunction with the RgParam operation for parsing answers to yes/no options of Executive parameters.

NlsYesNoOrBlank returns status code 13508 ("Not a yes or no answer") if the string given matches none of the strings in the yes or no lists.

Procedural Interface

NlsYesNoOrBlank (pNlsTableArea, pSdParam, pbCase): ercType

where

pNlsTableArea

NlsYesNoOrBlank returns a different value based on the following:

- If *pNlsTableArea* is 0 and [Sys]<Sys>Nls.sys was not present at system boot, an answer is returned based on the American strings "yes" and "no."

- If *pNlsTableArea* is 0 and [Sys]<Sys>Nls.sys was present at system boot, an answer is returned based on the NLS Yes or No Strings table.
- If *pNlsTableArea* is the memory address of an alternate set of NLS tables, an answer is returned based on the NLS Yes or No Strings table in the alternate set of NLS tables.

pSdParam

is the memory address of a six-byte block of memory. The first four bytes are the memory address of the string to be compared. The last two bytes contain the size of the string.

pbCase

is the memory address of a case value returned. Case values and their meanings are:

Case	**Meaning**
0	SdParam matches any of the list of nationalized words meaning no.
1	SdParam matches any of the list of the nationalized words meaning yes.
2	SdParam is the null string.

Request Block

NlsYesNoOrBlank is an object module procedure.

NlsYesOrNo (pNlsTableArea, pSdParam, pfNlsYNRet): ercType

Description

NlsYesOrNo performs a case insensitive string comparison against nationalized words meaning "yes" and "no."

The string comparison will complete successfully if the string passed matches any portion of a yes or no word. For example, in an American system, "y," "ye," and "yes" will match "yes."

It is recommended that this operation (or the related NlsYesNoOrBlank) be used in conjunction with the RgParam operation for parsing answers to yes/no options of Executive parameters.

If the size portion of the string parameter, sdParam.cb, is zero, NlsYesOrNo will return FALSE (that is, a match of "no").

NlsYesOrNo returns status code 13508 ("Not a yes or no answer") if the string given matches none of the strings in the yes or no lists.

Procedural Interface

NlsYesOrNo (pNlsTableArea, pSdParam, pfNlsYNRet): ercType

where

pNlsTableArea

NlsYesOrNo returns a different value based on the following:

- If *pNlsTableArea* is 0 and [Sys]<Sys>Nls.sys was not present at system boot, an answer is returned based on the American strings "yes" and "no."

- If *pNlsTableArea* is 0 and [Sys]<Sys>Nls.sys was present at system boot, an answer is returned based on the NLS Yes or No Strings table.
- If *pNlsTableArea* is address of an alternate set of NLS tables, an answer is returned based on the NLS Yes or No Strings table table in the alternate set of NLS tables.

pSdParam

is the memory address of a six-byte block of memory. The first four bytes are the memory address of the string to be compared. The last two bytes contain the size of the string.

pfNlsYNRet

is the memory address of a flag returned. The flag is set TRUE if the *SdParam* matches any of the list of international words meaning "yes." The flag is FALSE if *SdParam* matches any of the list of the international words meaning "no" or if *SdParam* indicates a null string.

Request Block

NlsYesOrNo is an object module procedure.

NPrint (pbString, cbString)

Description

NPrint prints the specified string to the video or other device. The NPrint and PutChar operations in CTOS.lib print to video byte streams [VID], using CheckErc for error handling. The NPrint and PutChar operations are used by the other print operations listed below:

OutputQuad	OutputWord
PutByte	PutPointer
PutQuad	PutWord
SbPrint	ZPrint

NPrint and PutChar can be replaced in a user program with public operations of the same names that do logging, different error handling, or redirection.

If a list file was opened using the OpenListFile operation, NPrint writes the string to the file as well as to the video device.

Procedural Interface

NPrint (pbString, cbString)

where

pbString
cbString

describe the character string to be written.

Request Block

NPrint is an object module procedure.

This page intentionally left blank

ObtainAccessInfo (pbNode, cbNode, pBuffer, sBuffer, pbCmdName, cbCmdName, pAccessCodeRet): ercType

Description

ObtainAccessInfo informs the application whether the current logged-on user name is allowed to use a specific Executive command (for example, **Cluster View** or **Set Time**) that executes on the specified node.

ObtainAccessInfo always determines access rights for the current user of the system on which the calling application is running; the application, therefore, does not need to specify a user name.

The application does, however, provide the name of the node that executes the command. In addition, the access file {Node}[Sys]<Sys>UserCmdConfig.sys must be present on this node. This access file must contain both the user's name and the name of the specified command.

Once called, ObtainAccessInfo in turn calls the Command Access Service. Subsequently, the Command Access Service examines the file {Node}[Sys]<Sys>UserCmdConfig.sys to determine allowable commands. (For details on the Command Access Service, see the *CTOS System Administrator's Guide*.)

ObtainAccessInfo is similar to ObtainUserAccessInfo, with one exception: ObtainUserAccessInfo determines access rights for any specified user, whereas ObtainAccessInfo determines access rights for the current user of the local system. (See ObtainUserAccessInfo.)

Status code 33 ("Command Access Service not installed") is returned if the Command Access Service is not currently installed. Status code 219 ("Command restricted") is returned if the user name cannot use the specified command.

Procedural Interface

ObtainAccessInfo (pbNode, cbNode, pBuffer, sBuffer, pbCmdName, cbCmdName, pAccessCodeRet): ercType

where

pbNode
cbNode

describe the node that executes the specified command.

pBuffer
sBuffer

describe a work area at least 100 bytes in length.

pbCmdName
cbCmdName

describe the command for which to return access information.

pAccessCodeRet

is the memory address of the 2-byte access code to be returned. Each access code indicates the following:

Code	**Description**
0	The specified user name was not found in the access file.
4	Only the specified user name was found in the access file.
5	The specified user name was found in the access file; the command is allowed.
6	The specified user name was found in the access file; the command is restricted.

Request Block

ObtainAccessInfo is an object module procedure.

This page intentionally left blank

ObtainUserAccessInfo (pbNode, cbNode, pBuffer, sBuffer, pbCmdName, cbCmdName, pbUser, cbUser, pAccessCodeRet): ercType

Description

ObtainUserAccessInfo informs the application whether any specified user name is allowed to use a given Executive command (for example, **Cluster View** or **Set Time**) that executes on the specified node.

The access file [Sys]<Sys>UserCmdConfig.sys must be present on the node that executes the command. In addition, this file must contain the specified user and command names.

Once called, ObtainUserAccessInfo in turn calls the Command Access Service. Subsequently, the Command Access Service examines the file {Node}[Sys]<Sys>UserCmdConfig.sys to determine allowable commands. (For details on the Command Access Service, see the *CTOS System Administrator's Guide*.)

ObtainUserAccessInfo is similar to ObtainAccessInfo, with one exception: ObtainAccessInfo determines access rights for the current user of the local system, whereas ObtainUserAccessInfo determines access rights for any specified user. (See ObtainAccessInfo.)

Status code 33 ("Command Access Service not installed") is returned if the Command Access Service is not installed. Status code 219 ("Command restricted") is returned if the user name cannot use the specified command.

Procedural Interface

ObtainUserAccessInfo (pbNode, cbNode, pBuffer, sBuffer, pbCmdName, cbCmdName, pbUser, cbUser, pAccessCodeRet): ercType

where

pbNode
cbNode

describe the node which executes the specified command.

pBuffer
sBuffer

describe a work area at least 100 bytes in length.

pbCmdName
cbCmdName

describe the command for which to return access information.

pbUser
cbUser

describe the user whose command access is to be determined.

pAccessCodeRet

is the memory address of a word to which the access code is to be returned. Each access code value indicates the following:

Code	**Description**
0	The specified user name was not found in the access file.
4	Only the specified user name was found in the access file.
5	The specified user name was found in the access file; the command is allowed.
6	The specified user name was found in the access file; the command is restricted.

Request Block

ObtainUserAccessInfo is an object module procedure.

This page intentionally left blank

OldReadHardId (pId): ercType

Caution: *This operation is similar to ReadHardId; however, it should be used by applications that need to run on older BTOS operating systems.*

Description

OldReadHardId retrieves an ID value (1 to 126) from either an add-on hardware ID device on an X-Bus workstation or the non-volatile RAM on an X-Bus or X-Bus+ workstation. The value of the hardware ID is saved even if the workstation is powered off.

On X-Bus workstations, if there is no physical hardware ID device, status code 693 ("No device present on the I-bus") is returned.

If it is installed, this call uses the new request code (8192); otherwise, it tries to use the old BTOS request code (49224).

Procedural Interface

OldReadHardId (pId): ercType

where

pId

is the memory address of a word to which the hardware ID is returned.

Request Block

OldReadHardId is an object module procedure.

This page intentionally left blank.

OldWriteHardId (wID): ercType

Caution: *This operation is similar to WriteHardId; however, it should be used by applications that need to run on older BTOS operating systems.*

Description

OldWriteHardId stores an ID value (1 to 126) for either an add-on hardware ID device on an X-Bus workstation or the non-volatile RAM on an X-Bus or X-Bus+ workstation. The value of the hardware ID is saved even if the workstation is powered off.

On an X-Bus workstation, if there is no physical hardware ID device, status code 693 ("No device present on the I-bus") is returned.

If it is installed, this call uses the new request code (8193); otherwise, it tries to use the old BTOS request code (49225).

Procedural Interface

OldWriteHardId (wID): ercType

where

wId

is a word value indicating the new hardware ID for the workstation.

Request Block

OldWriteHardId is an object module procedure.

This page intentionally left blank.

OpenByteStream (pBswa, pbFileSpec, cbFileSpec, pbPassword, cbPassword, mode, pBufferArea, sBufferArea): ercType

Description

OpenByteStream opens a device/file as a byte stream. If an output byte stream is opened for a file that does not already exist, OpenByteStream creates it. The address of the Byte Stream Work Area supplied to OpenByteStream must be supplied to subsequent operations such as ReadBytes, WriteBsRecord, and CloseByteStream to identify this particular byte stream.

Procedural Interface

OpenByteStream (pBswa, pbFileSpec, cbFileSpec, pbPassword, cbPassword, mode, pBufferArea, sBufferArea): ercType

where

pBswa

is the memory address of a 130-byte memory work area for use by SAM operations.

pbFileSpec
cbFileSpec

describe a device or file specification.

pbPassword
cbPassword

describe a device, volume, directory, or file password. The [Kbd], [Vid], [Comm], and [Nul] devices do not require passwords.

mode

is text, read, write, append, or modify. The mode is is indicated by 16-bit values representing the ASCII constants "mr", "mt", "mw", "ma", or "mm". In these ASCII constants, the first character (m) is the high-order byte and the second character (r, t, w, a, or m, respectively) is the low-order byte.

Text mode reads an existing file from the beginning to the end. The exception is word processed files. The text of word processed files is followed by formatting information. When text mode is specified, status code 1 ("End of file") is returned after the last byte of text is read, and the formatting information is ignored.

Read mode reads an existing file from the beginning to the end, including the formatting information at the end of word processed files.

Write mode overwrites a previously existing file of the specified name (if any) and adjusts the length as necessary. If a file of the specified name does not exist, SAM creates one.

Append mode appends output to the end of an existing file (if any). If a file of the specified name does not exist, SAM creates one.

Modify mode combines read mode and write mode, allowing both reading from and writing to the same device or file. The caller can use the GetBsLfa operation to locate the end of the file to append to it.

pBufferArea
sBufferArea

describe a memory area provided for the exclusive use of SAM operations. To ensure device independence, this area must be at least 1024 bytes and word-aligned. Providing a larger area improves the efficiency of file operations.

Request Block

OpenByteStream is an object module procedure.

OpenByteStreamC (pBs, pbFileSpec, cbFileSpec, pbPassword, cbPassword, mode, pBufferArea, sBufferArea): ercType

Description

OpenByteStreamC is the version of OpenByteStream that is device-specific to RS-232 serial ports ("[Comm]" and "[Ptr]" byte streams). (See "Device-Dependent SAM" and "Communications Programming" in the *CTOS Operating System Concepts Manual* for more information.)

Procedural Interface

OpenByteStreamC (pBs, pbFileSpec, cbFileSpec, pbPassword, cbPassword, mode, pBufferArea, sBufferArea): ercType

where

pBs

is the memory address of the Byte Stream Work Area (BSWA). Note that for Standard Software versions 12.1 or later, 160 bytes are required for this BSWA. For versions 12.0 or earlier, this BSWA requires 130 bytes.

pbFileSpec
cbFileSpec

describe the [COMM] or [PTR] device specification and configuration file, for example [PTRA]&PtrAConfig.sys.

pbPassword
cbPassword

describe a password needed to access the configuration file.

mode

is a code with one of the following values:

mr or mt	read mode
mw or ma	write mode
mm	modify mode (read and write)

pBufferArea
sBufferArea

describe the buffer that SamC will use. *sBufferArea* must be at least 512 bytes. (To circumvent buffer size restrictions, see Acquire-ByteStreamC.)

Request Block

OpenByteStreamC is an object module procedure.

OpenDaFile (pDawa, pbFilespec, cbFilespec, pbPassword, cbPassword, mode, pBuffer, sBuffer, sRecord): ercType

Description

OpenDaFile opens a DAM file in either read (shared) or modify (exclusive) mode. If the file does not exist, it is created. The address of the Direct Access Work Area supplied to OpenDaFile must be supplied to subsequent DAM operations.

Access to a DAM file is most efficient if its sectors are physically contiguous. This contiguity can be increased by preallocating the file. To preallocate the file, follow the call to OpenDaFile that creates the file with a call to WriteDaRecord. This call to WriteDaRecord should specify a value of *qiRecord* large enough to preallocate the desired file length. This "end record" can then be deleted.

Procedural Interface

OpenDaFile (pDawa, pbFilespec, cbFilespec, pbPassword, cbPassword, mode, pBuffer, sBuffer, sRecord): ercType

where

pDawa

is the memory address of a 64-byte memory work area for use by the Direct Access Method operations.

pbFilespec
cbFilespec

describe a character string specifying the name of the file to be opened.

pbPassword
cbPassword

describe a character string specifying a password that authorizes the requested file access.

mode

is read (shared) or modify (exclusive). The mode is indicated by 16-bit values representing the ASCII constants "mr" (mode read) or "mm" (mode modify). In these ASCII constants, the first character (m) is the high-order byte and the second character is the low-order byte.

pBuffer
sBuffer

describe a word-aligned memory area provided for the exclusive use of the Direct Access Method operations. *sBuffer* must be a multiple of 512 and must be greater or equal to *sRecord* plus 519. (This size constraint is relaxed in two cases. For details, see "Direct Access Method" in the *CTOS Operating System Concepts Manual.*)

sRecord

describes the fixed record size for the DAM file. If the DAM file already exists, *sRecord* must match the record size specified when the file was created.

Request Block

OpenDaFile is an object module procedure.

OpenFile (pFhRet, pbFileSpec, cbFileSpec, pbPassword, cbPassword, mode): ercType

Description

OpenFile opens an already existing file and returns a file handle. The file handle returned by OpenFile is used to refer to the file in subsequent operations such as Read, Write, and DeleteFile. OpenFile differs from ReopenFile in that each time a file is open with OpenFile, a new file handle is returned.

Procedural Interface

OpenFile (pFhRet, pbFileSpec, cbFileSpec, pbPassword, cbPassword, mode): ercType

where

pFhRet

is the memory address of the word where the file handle is returned.

pbFileSpec
cbFileSpec

describe a character string of the following form: {Node}[VolName]<DirName>FileName. The distinction between uppercase and lowercase in file specifications is not significant in matching file names.

pbPassword
cbPassword

describe the volume, directory, or file password that authorizes access to the specified file.

mode

is read, modify, or peek. This is indicated by 16-bit values representing the ASCII constants "mr" (mode read), "mm" (mode modify) or "mp" (mode peek). In these ASCII constants, the first character (m) is the high-order byte and the second character is the low-order byte.

Access in read mode permits the returned file handle to be used as an argument only to the CloseFile, CheckReadAsync, Read, ReadAsync, GetFhLongevity, GetFileStatus, and SetFhLongevity operations.

Access in modify mode, however, permits the returned file handle to be used as an argument to all operations that expect a file handle.

If the file is currently open in read mode, access in read mode is permitted but attempted access in modify mode causes the return of status code 220 ("File in use").

If the file is currently open in modify mode, attempted access in either read or modify mode causes the return of status code 220 ("File in use").

If the file is currently open in peek mode, access in either read or modify mode is permitted. If, however, the program that originally opened the file in peek mode attempts to read or to modify the file after it has been modified by another program, status code 210 ("Bad file handle") is returned.

Request Block

sFhMax is always 2.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	2
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	4
12	reserved	2	
14	mode	2	
16	reserved	2	
18	pbFileSpec	4	
22	cbFileSpec	2	
24	pbPassword	4	
28	cbPassword	2	
30	pFhRet	4	
34	sFhMax	2	2

This page intentionally left blank

OpenFileLL (pFhRet, pbFileSpec, cbFileSpec, pbPassword, cbPassword, mode): ercType

Description

OpenFileLL opens an already existing file and returns a file handle. The file handle is marked long-lived and can therefore be closed by the CloseFile and CloseAllFilesLL operations, but not by the CloseAllFiles operation or by program termination. The file handle returned by OpenFileLL is used to refer to the file in subsequent operations such as Read, Write, and DeleteFile.

Procedural Interface

OpenFileLL (pFhRet, pbFileSpec, cbFileSpec, pbPassword, cbPassword, mode): ercType

where

pFhRet

is the memory address of the word where the file handle is returned.

pbFileSpec
cbFileSpec

describe a character string of the following form: {Node}[VolName]<DirName>FileName. The distinction between uppercase and lowercase in file specifications is not significant in matching file names.

pbPassword
cbPassword

describe the volume, directory, or file password that authorizes access to the specified file.

mode

is read, modify, or peek. The mode is indicated by 16-bit values representing the ASCII constants "mr" (mode read), "mm" (mode modify) or "mp" (mode peek). In these ASCII constants, the first character (m) is the high-order byte and the second character is the low-order byte.

Access in read mode permits the returned file handle to be used as an argument only to the CloseFile, CheckReadAsync, Read, ReadAsync, GetFhLongevity, GetFileStatus, and SetFhLongevity operations. Access in modify mode, however, permits the returned file handle to be used as an argument to all operations that expect a file handle.

If the file is currently open in read mode, access in read mode is permitted but attempted access in modify mode causes the return of status code 220 ("File in use").

If the file is currently open in modify mode, attempted access in either read or modify mode causes the return of status code 220 ("File in use").

If the file is currently open in peek mode, access in either read or modify mode is permitted. If, however, the program that originally opened the file in peek mode attempts to read or to modify the file after it has been modified by another program, status code 210 ("Bad file handle") is returned.

Request Block

sFhMax is always 2.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	2
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	97
12	reserved	2	
14	mode	2	
16	reserved	2	
18	pbFileSpec	4	
22	cbFileSpec	2	
24	pbPassword	4	
28	cbPassword	2	
30	pFhRet	4	
34	sFhMax	2	2

This page intentionally left blank

OpenNlsFile (pbFileName, cbFileName, pbPassword, cbPassword, ppNlsTableRet): ercType

Description

OpenNlsFile opens the Native Language Support (NLS) file specified by *pbFileName/cbFileName*, loads the NLS tables contained in the file into short-lived memory, and returns the memory address of the tables. (For details on NLS, see "Native Language Support," in the *CTOS Operating System Concepts Manual.)* When this operation opens the file, it determines the amount of short-lived memory to allocate for the NLS tables. Then, it reads the tables from disk into the allocated space.

This operation is useful for accessing an NLS table area when multiple NLS table areas are being used or when an application is running in an environment in which the operating system has not loaded a default Nls.sys file.

Procedural Interface

OpenNlsFile (pbFileName, cbFileName, pbPassword, cbPassword, ppNlsTableRet): ercType

where

pbFileName
cbFileName

describe the file specification of the NLS file to be opened and loaded into short-lived memory. The file specification is a character string of the form *{Node}[VolName]<DirName>FileName.*

pbPassword
cbPassword

describe the volume, directory, or file password that authorizes access to the specified file.

ppNlsTableRet

is the memory address of a 4–byte memory area where the address of the NLS tables is returned.

Request Block

OpenNlsFile is an object module procedure.

OpenPSLogSession (pbDevName, cbDevName, wBlockID, wIterations, wLogHeapSize, pbShRet): ercType

Description

OpenPSLogSession a synonym for PSOpenLogSession. See the description of PSOpenLogSession for details.

This page intentionally left blank

OpenPSStatSession (pbDevName, cbDevName, pbBlockID, cbBlockID, pbShRet, cbShRet): ercType

Description

OpenPSStatSession is a synonym for PSOpenStatSession. See the description of PSOpenStatSession for details.

This page intentionally left blank

OpenRsFile (pRswa, pbFilespec, cbFilespec, pbPassword, cbPassword, mode, pBufferArea, sBufferArea): ercType

Description

OpenRsFile opens a Record Sequential Access Method file in read, write, or append mode. For write and append modes, if the file does not exist, it is created. The address of the Record Sequential Work Area supplied to OpenRsFile must be supplied to subsequent RSAM operations.

Procedural Interface

OpenRsFile (pRswa, pbFilespec, cbFilespec, pbPassword, cbPassword, mode, pBufferArea, sBufferArea): ercType

where

pRswa

is the memory address of a 150-byte memory work area for use by the Record Sequential Access Method operations.

pbFilespec
cbFilespec

describe a character string specifying the name of the file to be opened.

pbPassword
cbPassword

describe a character string specifying a password authorizing the requested file access.

mode

is read, write, or append. The mode is indicated by 16-bit values representing the ASCII constants "mr" (mode read), "mw" (mode write), or "ma" (mode append). In these ASCII constants, the first character (m) is the high-order byte and the second character (r, w, or a, respectively) is the low-order byte.

Mode read reads an existing file from the beginning.

Mode write overwrites a previously existing file of the specified name (if any) and adjusts the length as necessary. If a file of that name does not exist, RSAM creates one.

Mode append appends the records written to the end of an existing file (if any). If a file of the specified name does not exist, RSAM creates one.

pBufferArea
sBufferArea

describe a memory area provided for the exclusive use of the RSAM operations. This area must be at least 1,024 bytes long and word-aligned. Providing a larger area improves the efficiency of RSAM operations.

Request Block

OpenRsFile is an object module procedure.

OpenRtClock (pRqTime): ercType

Description

OpenRtClock establishes a Timer Request Block (TRB) between the client that requests the timing services and timer management. The client and timer management communicate by changing the fields in the TRB after an OpenRtClock call.

(See Chapter 4, "System Structures," for the format of a TRB.)

Procedural Interface

OpenRtClock (pRqTime): ercType

where

pRqTime

is the memory address of the client-provided TRB to be shared by the client and timer management.

Request Block

sRqTime is always 12.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	1
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	49
12	reserved	6	
18	pRqTime	4	
22	sRqTime	2	12

OpenServerMsgFile (pbFileSpec, cbFileSpec, pbPassword, cbPassword, fLLMemory): ercType

Description

OpenServerMsgFile is used to initialize a message file for use by a system service program or an application program that is using a relatively small message file. OpenServerMsgFile allocates memory and reads the entire contents of the given message file into that memory. Individual messages are then retrieved by calling GetServerMsg.

(See the related operations, GetServerMsg and CloseServerMsgFile.)

Procedural Interface

OpenServerMsgFile (pbFileSpec, cbFileSpec, pbPassword, cbPassword, fLLMemory): ercType

where

pbFileSpec
cbFileSpec

describe the name of the message file.

pbPassword
cbPassword

describe the password used in opening the message file.

fLLMemory

is TRUE or FALSE. TRUE implies that OpenServerMsgFile should allocate long-lived memory; FALSE implies that OpenServerMsgFile should allocate short-lived memory. Note that long-lived memory should not be used if a system service is opening the message file.

Request Block

OpenServerMsgFile is an object module procedure.

OpenTerminal has no procedural interface. You must make a request block and issue the request using Request, RequestDirect, or RequestRemote.

Caution: *This operation works on shared resource processor hardware only. Do not use it on workstation hardware.*

Description

OpenTerminal initiates the use of a specified port on either a Cluster or Terminal Processor. The returned terminal output buffer address is in linear format and is used for subsequent output operations. (See Chapter 4, "System Structures," for the format of the terminal output buffer.) In linear format, word ordering and byte ordering within each word are exactly the opposite to the Intel 80x86 processor convention, which places the most significant byte at the highest address.

On a terminal attached to an RS-232-C port, this request resets the terminal characteristics to the initial state specified in the configuration file and asserts Data Terminal Ready (DTR).

Procedural Interface

OpenTerminal has no procedural interface. You must make a request block and issue the request using one of the request operations listed above

where

bDestCPU

is the slot number of the Cluster or Terminal Processor to which the asynchronous terminal is connected.

bSourceCPU

is the slot number of the client.

bPort

is the port number of a terminal attached to an RS-232-C port.

bWindow

must be 0.

ppOutputBufferRet
spOutputBufferRet

refer to the output buffer for a terminal in the address space of *bDestCPU*. The returned memory address is in linear format. This linear pointer addresses the terminal output buffer. (See "Terminal Output Buffer" in Chapter 4, "System Structures.")

Request Block

sOutputBufferRet is always 4.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	4
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	1
4	userNum	2	
6	ExchResp	2	
8	ercRet	2	
10	rqCode	2	12306
12	bDestCPU	1	
13	bSourceCPU	1	
14	bPort	1	
15	bWindow	1	
16	ppOutputBufferRet	4	
20	spOutputBufferRet	2	4

This page intentionally left blank

OpenUserFile (pNlsTableArea, pFhRet, pBuffer, sBuffer, mode): ercType

Description

OpenUserFile opens an operator's .user file and returns the .user file handle. If the file is opened in modify mode, OpenUserFile first calls the CopyFile operation to copy the file to a -old file.

If a Native Language Support (NLS) table area is present at system boot, the file suffixes .user and -old can be nationalized. See the description of *pNlsTableArea.* (For details on NLS, see "Native Language Support" in the *CTOS Operating System Concepts Manual.*)

Procedural Interface

OpenUserFile (pNlsTableArea, pFhRet, pBuffer, sBuffer, mode): ercType

where

pNlsTableArea

The file suffixes *.user* and *-old* are constructed differently based on the following:

- If *pNlsTableArea* is 0 and [Sys]<Sys>Nls.sys was not present at system boot, the suffix for the user file is .user and the -old file is -old.
- If *pNlsTableArea* is 0 and [Sys]<Sys>Nls.sys was present at system boot, the suffixes are constructed based upon the values in the NLS Strings table.
- If *pNlsTableArea* is the memory address of an alternate set of NLS tables, the suffixes are constructed based upon the values in the NLS Strings table in the alternate set of NLS tables.

pFhRet

is a memory address of the word where the .user file handle is returned.

pBuffer
sBuffer

describe a buffer to be used in modify mode for copying the existing contents of the user file to a -old file. *sBuffer* must be at least 512 bytes. In the user file is opened in read mode, both of these parameters should be 0.

mode

is a word that specifies read, modify, or peek. *mode* is indicated by 16-bit values representing the ASCII constants "mr" (mode read), "mm" (mode modify), or "mp" (mode peek). In these ASCII constants, the first character (m) is the high-order byte, and the second character is the low-order byte.

The .user file is saved only if the file is opened in modify mode.

Request Block

OpenUserFile is an object module procedure.

OpenUserFile (pNlsTableArea, pFhRet, pBuffer, sBuffer, mode): ercType

Description

OpenUserFile opens an operator's .user file and returns the .user file handle. If the file is opened in modify mode, OpenUserFile first calls the CopyFile operation to copy the file to a -old file.

If a Native Language Support (NLS) table area is present at system boot, the file suffixes .user and -old can be nationalized. See the description of *pNlsTableArea.* (For details on NLS, see "Native Language Support" in the *CTOS Operating System Concepts Manual.*)

Procedural Interface

OpenUserFile (pNlsTableArea, pFhRet, pBuffer, sBuffer, mode): ercType

where

pNlsTableArea

The file suffixes *.user* and *−old* are constructed differently based on the following:

- If *pNlsTableArea* is NIL and [Sys]<Sys>Nls.sys was not present at system boot, the suffix for the user file is .user and the −old file is -old.
- If *pNlsTableArea* is NIL and [Sys]<Sys>Nls.sys was present at system boot, the suffixes are constructed based upon the values in the NLS Strings table.
- If *pNlsTableArea* is the memory address of an alternate set of NLS tables, the suffixes are constructed based upon the values in the NLS Strings table in the alternate set of NLS tables.

pFhRet

is a memory address of the word where the .user file handle is returned.

pBuffer
sBuffer

describe a buffer to be used in modify mode for copying the existing contents of the user file to a -old file. *sBuffer* must be at least 512 bytes. In the user file is opened in read mode, both of these parameters should be 0.

mode

is a word that specifies read, modify, or peek. *mode* is indicated by 16-bit values representing the ASCII constants "mr" (mode read), "mm" (mode modify), or "mp" (mode peek). In these ASCII constants, the first character (m) is the high-order byte, and the second character is the low-order byte.

The .user file is saved only if the file is opened in modify mode.

Request Block

OpenUserFile is an object module procedure.

OpenVidFilter: ercType

Description

OpenVidFilter sets a flag in the Application System Control Block (ASCB) to redirect the video byte stream output of a program(s) to which the caller chains. The video output is written to the temporary file [Scr]<$>bsFilter.tmp rather than being displayed to the screen. With each successive write, the output is appended to the file.

To provide a buffer for the redirected byte stream, the program whose output is filtered must explicitly open a video byte stream (call the OpenByteStream operation, specifying *bsVid* as the device). (See the description of OpenByteStream for details.) The program should also close the byte stream using the CloseByteStream operation after all output is written to the file.

When the program whose output is filtered exits, the caller of OpenVidVilter must call the CloseVidFilter operation to clear the flag in the ASCB. CloseVidFilter also closes the byte stream (if closing was not already done).

The Batch program uses the OpenVidFilter and CloseVidFilter operations to suppress video messages in programs it runs.

Procedural Interface

OpenVidFilter: ercType

Request Block

OpenVidFilter is an object module procedure.

This page intentionally left blank

OsVersion (pVersionRet): ercType

Description

OsVersion is used by programs to determine the version of an operating system.

The version fields are a system build parameter.

NOTE: To avoid system crashes with status code 31 ("No such request code") when using the request procedural interface, programs that may run on earlier versions of the operating system should use the library procedure CurrentOsVersion rather than this request.

Procedural Interface

OsVersion (pVersionRet): ercType

where

pVersionRet

is the memory address of a word to which the following operating system release and revision numbers are returned:

Offset	Field	Size (Bytes)
0	bRevision	1
1	bRelease	1

For example, for operating system version 9.1, bRevision would be 1 and bRelease would be 9.

The operating system internal version numbers and their corresponding release numbers are listed below.

NOTE: BTOS II real mode operating systems have an internal version number of 9.8 or 9.11.

CTOS and BTOS Operating Systems:

Internal Version Number	**Release Numbers**		
	CTOS and SRP	**XEBTOS**	**BTOS**
8.0	CTOS 8.0		BTOS 0–3.1
9.0	CTOS 9.0	XEBTOS 6.0	BTOS 4.0
9.1	CTOS 9.1		BTOS 5.0
9.5	CTOS 9.5		BTOS 6.0
9.6	CTOS 9.6		BTOS 7.0
9.7	CTOS 9.7		BTOS 8.0
9.8	CTOS 9.8		real mode BTOS II (versions less than 3.2)
9.9	CTOS 9.9		
9.10	CTOS 9.10		
9.11			real mode BTOS II 3.2
10.0	CTOS II 1.0		
10.4	CTOS/SRP 1.4	XEBTOS 7.0	
11.0	CTOS/VM 2.0		BTOS II 1.0
11.1	CTOS/VM 2.1		BTOS II 1.1
11.2	CTOS/VM 2.2		
11.3	CTOS/VM 2.3		
11.4			BTOS II 3.0
11.5			BTOS II 3.1
11.6	CTOS/VM 2.4		
11.7			protected mode BTOS II 3.2
12.0	CTOS/XE 3.0		BTOS II 2.0.4

Merged Operating Systems:

Internal Version Number	Release Number
9.13	CTOS I 3.3
12.0	CTOS/XE 3.0
12.3	CTOS II 3.3
9.14	CTOS I 3.4
12.4	CTOS II 3.4
12.1	CTOS/XE 3.4
13.0	CTOS III 1.0

Request Block

sVersionRet is always 2.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	0
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	261
12	pVersionRet	4	
16	sVersionRet	2	2

This page intentionally left blank.

OutputBytesWithWrap (pbString, cbString): ercType

Description

OutputBytesWithWrap is used to output a string to the video byte stream. The string is broken up into words such that, if the current word will not fit on the current line, the routine outputs a carriage return and a tab before proceeding with rest of the message. A word is defined as a string of characters up to a character that has a value of 20h or less.

The Executive uses this routine to print messages, such as those displayed while copying, renaming, and deleting.

Procedural Interface

OutputBytesWithWrap (pbString, cbString): ercType

where

pbString

is the memory address of the string to be written.

cbString

is the count of bytes to write.

Request Block

OutputBytesWithWrap is an object module procedure.

This page intentionally left blank

OutputQuad (q, base, field, bPad)

NOTE: This operation does not return a status code.

Description

OutputQuad prints a quad (32-bit unsigned integer) to the video or other device as specified with the NPrint and PutChar operations.

The operation allows the program to specify the base, field width, and pad character.

Procedural Interface

OutputQuad (q, base, field, bPad)

where

q

is a 32-bit number to be printed.

base

is the base from 2 to 16 in which the number is to be displayed, such as 10 (decimal) or 16 (hex).

field

is the minimum number of characters to be used for the display. The larger of this value and the actual number of characters required for the number is used.

bPad

is the character to be used for padding if the field above is larger than the number of characters needed to display the number.

Request Block

OutputQuad is an object module procedure.

OutputToVid0 (pbString, cbString): ercType

Description

OutputToVid0 allows a program to perform minimal output to the video device [VID] while optionally excluding video byte streams. This call is typically used by a system service that only wants to output some appropriate installation message or an error message if the installation did not proceed correctly.

OutputToVid0 is a minimal set of video byte stream code. It supports sequential characters sent to the video device. The carriage return character (10h) is the only character interpreted in a special way.

When compared to the amount of code required to link with the full video byte stream and use operations such as WriteBsRecord, OutputToVid0 adds significantly less code to your program.

If you want to leave out video byte streams and need instructions on how to do so, see the documentation on building a customized SamGen in the *CTOS Programming Guide.*

Procedural Interface

OutputToVid0 (pbString, cbString): ercType

where

pbString
cbString

describe the string to be displayed.

Request Block

OutputToVid0 is an object module procedure.

OutputWord (w, base, field, bPad)

NOTE: This operation does not return a status code.

Description

OutputWord prints a word (16-bit unsigned integer) to the video or other device as specified by the NPrint and PutChar operations.

The operation allows the program to specify the base, field width, and pad character.

Procedural Interface

OutputWord (w, base, field, bPad)

where

w

is a 16-bit number to be printed.

base

is the base from 2 to 16 in which the number is to be displayed, such as 10 (decimal) or 16 (hex).

field

is the minimum number of characters to be used for the display. The larger of this value and the actual number of characters required for the number is used.

bPad

is the character to be used for padding if the field above is larger than the number of characters needed to display the number.

Request Block

OutputWord is an object module procedure.

NOTE: PaFromP is an alias for LaFromP. See LaFromP.

Deleted with Update Notice 1, August, 1992.

NOTE: PaFromSn is an alias for LaFromP. See LaFromSn.

Deleted with Update Notice 1, August, 1992.

ParseFileSpec (userNum, pbInputSpec, cbInputSpec, fDefaultPath, pbNode, pcbNode, pbVol, pcbVol,pbDir, pcbDir, pbFile, pcbFile,pbPassword, pcbPassword,fCanonical, bSpecType): word

Description

ParseFileSpec searches a file specification and returns each *token* (that is, node, volume, directory, or file name string or password string) and the token size for each level specified. The *levels* are node, volume, directory, file name, and password.

With each level parsed, ParseFileSpec performs a validity check and returns a word value (rather than an ercType). If no errors are detected, a value of 0 is returned. Otherwise, a nonzero value is returned. (For details, see "File Management" in the *CTOS Operating System Concepts Manual*. The meanings of the nonzero values returned are listed in Table 11-3.)

Say, for example, the caller elects to parse a full file specification, that is, one containing the levels

{Node}
[Volume]
<Dir>
FileName
^Password

For each level, the caller provides the memory address of a buffer to which the token for that level will be returned and the address of a word to which the token size will be returned.

NOTE: It is up to the caller to ensure that adequate memory is allocated for buffers to contain the maximum size tokens that can be returned.

If the caller then provides the memory address of the following specification string and its size (26) for *pbInputSpec* and *cbInputSpec*, respectively,

{Local}[Sys]<Dir1>Foo^1234

ParseFileSpec removes the brackets and returns the following tokens and their byte counts to the buffers located at the specified addresses:

Token	**Byte Count**
Local	5
Sys	3
Dir1	4
Foo	3
1234	4

If any memory addresses of buffers and byte counts are 0, ParseFileSpec interprets such addresses as meaning unspecified levels, and no attempt is made to parse them.

If, however, the addresses are not 0 and the levels are not provided in the input file specification, the corresponding default path tokens are returned, provided the flag *fDefaultPath* is set to TRUE.

To parse each level of the file specification, ParseFileSpec calls the following lower layer procedures:

ParseSpecForNode
ParseSpecForVol
ParseSpecForDir
ParseSpecForFile
ParseSpecForPassword

If your application only requires parsing a subset of the levels (one or two, for example), your application can call any of these procedures individually. For details, see the descriptions of these procedures in this chapter. There is also a discussion on accessing the file parsing procedures at different levels in "File Management" in the *CTOS Operating System Concepts Manual.*

Procedural Interface

ParseFileSpec (userNum, pbInputSpec, cbInputSpec, fDefaultPath, pbNode, pcbNode, pbVol, pcbVol,pbDir, pcbDir, pbFile, pcbFile,pbPassword, pcbPassword,fCanonical, bSpecType): word

where

userNum

is the user number (word). The default is 0 for the calling program.

pbInputSpec
cbInputSpec

describe the input file specification to parse. The size of the specification includes all brackets and tokens.

fDefaultPath

is a flag (byte) with one of the following values:

Value	**Meaning**
TRUE	If the caller requests that a level be parsed that does not exist in the input file specification, the corresponding token from the default path as specified in the User Control Block is returned.
FALSE	If the caller requests that a level be parsed that does not exist in the input file specification (the corresponding token from the default path is not returned), a zer length string is returned.

pbNode

is the memory address of the buffer to which the node token is returned or is 0.

pcbNode

is the memory address of a word to which the node token size is returned or is 0.

pbVol

is the memory address of the buffer to which the volume token is returned or is 0.

pcbVol

is the memory address of a word to which the volume token size is returned or is 0.

pbDir

is the memory address of the buffer to which the directory token is returned or is 0.

pcbDir

is the memory address of a word to which the directory token size is returned or is 0.

pbFile

is the memory address of the buffer to which the file name token is returned or is 0.

pcbFile

is the memory address of a word to which the file name token size is returned or is 0.

pbPassword

is the memory address of the buffer to which the password token is returned or is 0.

pcbPassword

is the memory address of a word to which the password token size is returned or is NIL.

fCanonical

is a flag (byte). If *fCanonical* is TRUE, the canonical form of the node and/or volume name is returned.

bSpecType

is one of the following values:

Value	**Meaning**
0	Treat the token without brackets as the file name. For example, foo is the file name in the specification {Local}[Sys]<Dir>foo
1	Treat the token without brackets as the volume name. For example, foo is the volume name in the specification {Local}foo
2	Treat the token without brackets as the directory name. For example, foo is the directory name in the specification [Sys]foo

Request Block

ParseFileSpec is an object module procedure.

This page intentionally left blank

ParseSpecForDir (pbInputSpec, cbInputSpec, pbDefaultDir, cbDefaultDir, pbDir, pcbDir, bSpecType): word

Description

ParseSpecForDir searches a file specification to find the directory level, removes the brackets at this level, and stores the directory token and its size at memory addresses provided by the caller.

ParseSpecForDir performs a validity check and returns a word value (rather than an ercType). If no errors are detected, a 0 value is returned. Otherwise, a nonzero value is returned. (For details, see "File Management" in the *CTOS Operating System Concepts Manual*.)

NOTE: It is up to the caller to ensure that adequate memory is allocated for the buffer to contain the maximum size token that can be returned.

Say, for example, the input file specification to this operation is

{Local}[Sys]<Dir1>Foo

If the memory addresss of the directory token buffer and the token size are not 0, the token Dir1 and the token size (4) are returned. Otherwise, the directory level is not parsed. (See the parameter descriptions of *pbDir* and *pcbDir*.)

The ParseFileSpec operation calls this procedure to parse and validate the directory level of a file specification. (For details, see the description of ParseFileSpec in this chapter.)

Procedural Interface

ParseSpecForDir (pbInputSpec, cbInputSpec, pbDefaultDir, cbDefaultDir, pbDir, pcbDir, bSpecType): word

where

pbInputSpec
cbInputSpec

describe the input file specification to parse. The size of the specification includes all brackets and tokens.

pbDefaultDir
cbDefaultDir

describe the current default directory and the directory size, or are zero values.

If the current default directory is described and the directory does not exist in the input file specification, the default directory token and its size are returned to the addresses *pbDir* and *pcbDir*, respectively. (See below.) If, however, *pbDefaultDir* and *cbDefaultDir* are 0, the default directory token is not returned.

pbDir
pcbDir

can be either of the following values:

Value	**Meaning**
zero pointers	The directory is not parsed and returned to the caller.
directory/directory size pointers	If the directory is present in the input file specification, the directory is parsed and returned to the caller. Otherwise, if *pbDefaultDir* and *cbDefaultDir* describe the default directory, the default directory token is returned.

bSpecType

is a byte with one of the following values:

Value	**Meaning**
0	Treat the token without brackets as the file name. For example, foo is the file name in the specification {Local}[Sys]<Dir>foo
2	Treat the token without brackets as the directory name. For example, foo is the directory name in the specification [Sys]foo

Request Block

ParseSpecForDir is an object module procedure.

This page intentionally left blank.

ParseSpecForFile (pbInputSpec, cbInputSpec, pbDefaultFile,cbDefaultFile, pbFile, pcbFile, bSpecType): word

Description

ParseSpecForFile searches a file specification to find the file name level and stores the file name token and its size at memory addresses provided by the caller.

ParseSpecForFile performs a validity check and returns a word value (rather than an ercType). If no errors are detected, a 0 value is returned. Otherwise, a nonzero value is returned. (For details, see "File Management" in the *CTOS Operating System Concepts Manual.*)

NOTE: It is up to the caller to ensure that adequate memory is allocated for the buffer to contain the maximum size token that can be returned.

Say, for example, the input file specification to this operation is

{Local}[Sys]<Dir1>Foo

If the memory addresss of the file name token buffer and the token size are not 0, the token Foo and the token size (3) are returned. Otherwise, the file name is not parsed. (See the parameter descriptions of *pbFile* and *pcbFile*.)

The ParseFileSpec operation calls this procedure to parse and validate the file name level of a file specification. (For details, see the description of ParseFileSpec in this chapter.)

Procedural Interface

ParseSpecForFile (pbInputSpec, cbInputSpec, pbDefaultFile, cbDefaultFile, pbFile, pcbFile, bSpecType): word

where

pbInputSpec
cbInputSpec

describe the input file specification to parse. The size of the specification includes all brackets and tokens.

pbDefaultFile
cbDefaultFile

are not used.

pbFile
pcbFile

can be either of the following values:

Value	**Description**
zero pointers	The file name is not parsed and returned to the caller.
file/file size pointers	The file name is parsed and returned to the caller.

bSpecType

is a byte value (currently ignored).

Request Block

ParseSpecForFile is an object module procedure.

ParseSpecForNode (pbInputSpec, cbInputSpec, pbDefaultNode, cbDefaultNode, pbNode, pcbNode, bSpecType): word

Description

ParseSpecForNode searches a file specification to find the node level, removes the brackets at this level, and stores the node token and its size at memory addresses provided by the caller.

ParseSpecForNode performs a validity check and returns a word value (rather than an ercType). If no errors are detected, a zero value is returned. Otherwise, a nonzero value is returned. (For details, see "File Management" in the *CTOS Operating System Concepts Manual.*)

NOTE: It is up to the caller to ensure that adequate memory is allocated for the buffer to contain the maximum size token that can be returned.

Say, for example, the input file specification to this operation is

{Local}[Sys]<Dir1>Foo

If the memory addresss of the node token buffer and the token size are not 0, the token Local and the token size (5) are returned. Otherwise, the node level is not parsed. (See the parameter descriptions of *pbVol* and *pcbVol.*)

The ParseFileSpec operation calls this procedure to parse and validate the node level of a file specification. (For details, see the description of ParseFileSpec in this chapter.)

Procedural Interface

ParseSpecForNode (pbInputSpec, cbInputSpec, pbDefaultNode, cbDefaultNode, pbNode, pcbNode, bSpecType): word

where

pbInputSpec
cbInputSpec

describe the input file specification to parse. The size of the specification includes all brackets and tokens.

pbDefaultNode
cbDefaultNode

describe the current default node and the node size, or are zero values.

If the current default node is described and the node does not exist in the input file specification, the default node token and its size are returned to the addresses *pbNode* and *pcbNode*, respectively. (See below.) If, however, *pbDefaultNode* and *cbDefaultNode* are 0, the default node token is not returned.

pbNode
pcbNode

can be either of the following values:

Value	**Meaning**
zero pointers	The node is not parsed and returned to the caller.
node/node size pointers	If the node is present in the input file specification, the node is parsed and returned to the caller. Otherwise, if *pbDefaultNode* and *cbDefaultNode* describe the default node, the default node token is returned.

bSpecType

is a byte value (currently ignored).

Request Block

ParseSpecForNode is an object module procedure.

This page intentionally left blank

ParseSpecForPassword (pbInputSpec, cbInputSpec, pbDefaultPassword, cbDefaultPassword, pbPassword, pcbPasword, bSpecType): word

Description

ParseSpecForPassword searches a file specification to find the password level, removes the caret(^), and stores the password token and its size at memory addresses provided by the caller.

ParseSpecForPassword checks for a valid password and returns a word value (rather than an ercType). If the password is valid, a zero value is returned. Otherwise, a nonzero value is returned. (For details, see "File Management" in the *CTOS Operating System Concepts Manual.*)

NOTE: It is up to the caller to ensure that adequate memory is allocated for the buffer to contain the maximum size token that can be returned.

Say, for example, the input file specification to this operation is

{Local}[Sys]<Dir1>Foo^1234

If the memory addresses of the password token buffer and the token size are not 0, the token 1234 and the token size (4) are returned. Otherwise, the password level is not parsed. (See the parameter descriptions of *pbPassword* and *pcbPassword.*)

The ParseFileSpec operation calls this procedure to parse and validate the password level of a file specification. (For details, see the description of ParseFileSpec in this chapter.)

Procedural Interface

ParseSpecForPassword (pbInputSpec, cbInputSpec, pbDefaultPassword, cbDefaultPassword, pbPassword, pcbPassword, bSpecType): word

where

pbInputSpec
cbInputSpec

describe the input file specification to parse. The size of the specification includes all brackets and tokens.

pbDefaultPassword
cbDefaultPassword

describe the current default password and the password size, or are zero values.

If the current default password is described and the password does not exist in the input file specification, the default password token and its size are returned to the addresses *pbPassword* and *pcbPassword*, respectively. (See *pbPassword* and *pcbPassword*, below.) If, however, *pbDefaultPassword* and *cbDefaultPassword* are 0, the default password token is not returned.

pbPassword
pcbPassword

can be either of the following values:

Value	Meaning
zero pointers	The password is not parsed or returned to the caller.
password/password size pointers	If the password is present in the input file specification, the password is parsed and returned to the caller. Otherwise, if *pbDefaultPassword* and *cbDefaultPassword* describe the default password, the default node token is returned.

bSpecType

is a byte value (currently ignored).

Request Block

ParseSpecForPassword is an object module procedure.

This page intentionally left blank.

ParseSpecForVol (pbInputSpec, cbInputSpec, pbDefaultVol, cbDefaultVol, pbVol, pcbVol, bSpecType): word

Description

ParseSpecForVol searches a file specification to find the volume level, removes the brackets at this level, and stores the volume token and its size at memory addresses provided by the caller.

ParseSpecForVol performs a validity check and returns a word value (rather than an ercType). If no errors are detected, a zero value is returned. Otherwise, a nonzero value is returned. (For details, see "File Management" in the *CTOS Operating System Concepts Manual*.)

NOTE: It is up to the caller to ensure that adequate memory is allocated for the buffer to contain the maximum size token that can be returned.

Say, for example, the input file specification to this operation is

{Local}[Sys]<Dir1>Foo

If the memory addresses of the volume token buffer and the token size are not 0, the token Sys and the token size (3) are returned. Otherwise, the volume level is not parsed. (See the parameter descriptions of *pbVol* and *pcbVol*.)

The ParseFileSpec operation calls this procedure to parse and validate the volume level of a file specification. (For details, see the description of ParseFileSpec in this chapter.)

Procedural Interface

ParseSpecForVol (pbInputSpec, cbInputSpec, pbDefaultVol, cbDefaultVol, pbVol, pcbVol, bSpecType): word

where

pbInputSpec
cbInputSpec

describe the input file specification to parse. The size of the specification includes all brackets and tokens.

pbDefaultVol
cbDefaultVol

describe the current default volume and the volume size, or are zero values.

If the current default volume is described and the volume does not exist in the input file specification, the default volume token and its size are returned to the addresses *pbVol* and *pcbVol*, respectively. (See below.) If, however, *pbDefaultVol* and *cbDefaultVol* are 0, the default volume token is not returned.

pbVol
pcbVol

can be either of the following values:

Value	**Meaning**
zero pointers	The volume is not parsed or returned to the caller.
volume/volume size pointers	If the volume is present in the input file specification, the volume is parsed and returned to the caller. Otherwise, if *pbDefaultVol* and *cbDefaultVol* describe the default volume, the default volume token is returned.

bSpecType

is a byte with one of the following values:

Value	**Meaning**
0	Treat the token without brackets as the file name. For example, foo is the file name in the specification {Local}[Sys]<Dir>foo
1	Treat the token without brackets as the volume name. For example, foo is the volume name in the specification {Local}foo

Request Block

ParseSpecForVol is an object module procedure.

This page intentionally left blank

ParseTime (psbDateTime, pExpDateTimeStruct): ercType

Description

ParseTime converts a caller-supplied string into the Expanded Date/Time format. (For details on the expanded format, see Chapter 4, "System Structures.")

The operation fills in only those fields in the Expanded Date/Time format that are found in the string, with the following exceptions:

- If any field in the Expanded Date/Time format is initialized to have a value of −1 (0FFh for 1–byte fields or 0FFFFh for 2–byte fields), an error is returned if there is no entry in the string for that field. If there is more than one entry in the string for a field, the last one is used.

- If the field *dayOfMonth* in the Expanded Date/Time format is initialized to have a value of 0, status code 3806 ("Day of month is out of range 0 to 31") is returned.

For ease in nationalization, programs should use the NlsParseTime operation rather than ParseTime. (For details, see "Native Language Support" in the *CTOS Operating System Concepts Manual.*)

Procedural Interface

ParseTime (psbDateTime, pExpDateTimeStruct): ercType

where

psbDateTime

is the memory address of a string containing the date/time. The first byte of the string is the size of the string; subsequent bytes contain the characters in the string.

pExpDateTimeStruct

is the memory address of the Expanded Date/Time format, as returned by the ExpandDateTime operation, and/or initialized to have −1 as the value of each field.

Request Block

ParseTime is an object module procedure.

PDAssignMouse (wUserNumber): ercType

Description

PDAssignMouse assigns the mouse (or any pointing device) to the specified user number. This operation is normally used by system programmers.

For details on Mouse Services, see the *CTOS Programming Guide.*

Procedural Interface

PDAssignMouse (wUserNumber): ercType

where

wUserNumber

specifies the user number to which the mouse is assigned.

Request Block

Offset	**Field**	**Size (Bytes)**	**Contents**
0	sCntInfo	1	2
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	32789
12	wUserNumber	2	

This page intentionally left blank

PdGetCursorPos (pwXPos, pwYPos): ercType

Description

PdGetCursorPos returns the current coordinates of the cursor in virtual screen coordinates. In most cases, the cursor position is returned by ReadInputEvent. Status code 4802 is returned if the application is not the current context.

See also ReadInputEvent, which is the commoly used operation to set the cursor position, and PdGetCursorPosNSC, which uses normalized screen coordinates.

For details on Mouse Services, see the *CTOS Programming Guide.*

Procedural Interface

PdGetCursorPos (pwXPos, pwYPos): ercType

where

pwXPos
pwYPos

are the memory addresses of the cursor coordinates.

Request Block

PdGetCursorPos is an object module procedure in Mouse.lib.

This page intentionally left blank

PDGetCursorPosNSC (pwXPos, pwYPos): ercType

Description

PDGetCursorPosNSC returns the current cursor coordinates in normalized screen coordinates. Normalized screen coordinates are generally used by the Context Manager. Status code 4802 is returned if the application is not the current context.

See also PDGetCursorPos, which uses virtual screen coordinates.

For details on Mouse Services, see the *CTOS Programming Guide*.

Procedural Interface

PDGetCursorPosNSC (pwXPos, pwYPos): ercType

where

pwXPos
pwYPos

are the memory addresses of the cursor coordinates.

Request Block

sXPos and *sYPos* are always 2.

Offset	Field	Size (Bytes)	Contents
0	sCntlInfo	1	6
1	rtInfo	1	0
2	nReqPbCb	1	
3	nRespPbCb	1	2
4	UserNum	2	
6	exchRet	2	
8	ercRet	2	
10	rqCode	2	32788
12	reserved	6	
18	pwXPos	4	
22	sXPos	2	2
24	pwYPos	4	
28	sYPos	2	2

PDInitialize: ercType

Description

PDInitialize resets the mouse routines. This operation should be used after the caller has changed screen format from 80- to 132-column format, or has changed the graphics resolution AND the caller is NOT using the PDSetVirtualCoordinates or the PDSetCharMapVirtualCoordinates operation. If the caller is using either of the preceding operations after the format change, there is no need to call PDInitialize, since this would have been done automatically.

For details on Mouse Services, see the *CTOS Programming Guide*.

Procedural Interface

PDInitialize: ercType

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntlInfo	1	0
1	rtInfo	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchRet	2	
8	ercRet	2	
10	rqCode	2	33170

This page intentionally left blank

PDLoadCursor (pCursorShape, sCursorShape, bType): ercType

Description

PDLoadCursor loads the cursor bitmap if a graphics cursor has been requested. (See the description of PDSetCursorType). When tracking is enabled, the picture in the bitmap is displayed and moved on the screen in response to movement of the mouse.

Procedural Interface

PDLoadCursor (pCursorShape, sCursorShape, bType): ercType

where

pCursorShape
sCursorShape

describe the memory area for an icon structure that defines the cursor shape. The format of the icon structure is shown below:

Offset	Field	Size (bytes)	Description
0	wSignature	2	IC'
2	wVersion	2	0
4	sIcon	2	Size of icon bitmap
6	cbLine	2	Number of bytes per line in icon
8	bWidth	1	Icon width in pixels
9	bHeight	1	Icon height in pixels

Offset	Field	Size (bytes)	Description
10	bxOffset	1	Icon hot spot x offset from left
11	byOffset	1	Icon hot spot y offset from top
12	bMaskFlag	1	If bMaskFlag is the value 1, it specifies that the icon has a mask*
13	rgbReserved	3	
16	sbName	16	Name of icon
32	rgwBitmap	sIcon	Icon bitmap

* = not implemented

bType

indicates the cursor type. The values of *bType* are

1	Mask overlaid (AND/OR)
2	Single overlaid (OR)
3	Exclusive OR (XOR)

If you design your own graphics cursor, note that the bits are flipped in the word when they are loaded into screen memory. In addition, the bit map must be an integral number of bytes.

For example, a left, upward-pointing arrow is shown in Figure 3-5. Figure 3-6 shows the first six rows of this icon loaded into memory, from right to left. Figure 3-7 shows the Pascal code defining this cursor. (Also see the example in "Mouse Services" in the *CTOS Programming Guide*.)

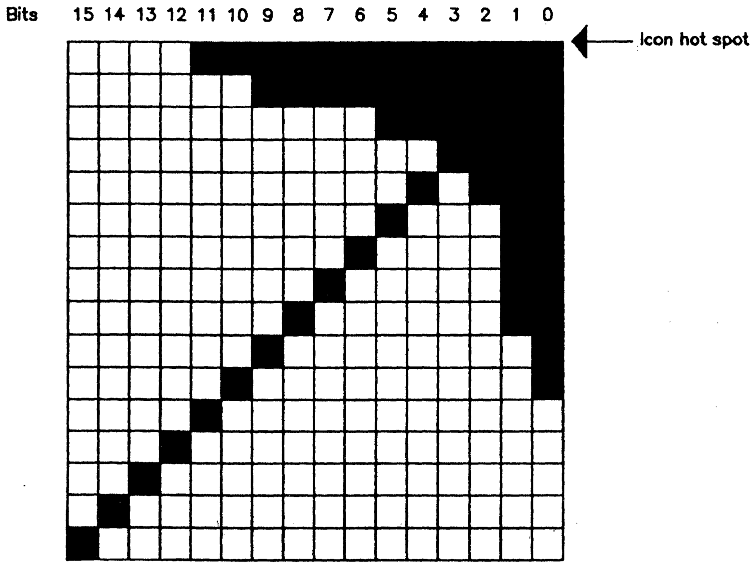


Figure 3-5. User-Defined Graphics Cursor

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0
1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0
1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0
1	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0

Figure 3-6. Six Rows of Sample Graphics Cursor Loaded into Screen Memory

```
DrawCursor.Pattern[0] := 16#0FFF;
DrawCursor.Pattern[1] := 16#03FF;
DrawCursor.Pattern[2] := 16#003FF;
DrawCursor.Pattern[3] := 16#000F;
DrawCursor.Pattern[4] := 16#0017;
DrawCursor.Pattern[5] := 16#0027;
DrawCursor.Pattern[6] := 16#0043;
DrawCursor.Pattern[7] := 16#0083;
DrawCursor.Pattern[8] := 16#0103;
DrawCursor.Pattern[9] := 16#0203;
DrawCursor.Pattern[10] := 16#0401;
DrawCursor.Pattern[11] := 16#0801;
DrawCursor.Pattern[12] := 16#1000;
DrawCursor.Pattern[13] := 16#2000;
DrawCursor.Pattern[14] := 16#4000;
DrawCursor.Pattern[15] := 16#8000
```

Figure 3-7. Pascal Code Defining the Left-Arrow Cursor.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntlInfo	1	6
1	rtInfo	1	0
2	nReqPbCb	1	1
3	nRespPbCb	1	0
4	UserNum	2	
6	exchRet	2	
8	ercRet	2	
10	rqCode	2	32790
12	bType	2	
14	reserved	4	
18	pCursorShape	4	
22	sCursorShape	2	

This page intentionally left blank

PDLoadSystemCursor (pCursorShape,sCursorShape, bType): ercType

Description

PDLoadSystemCursor loads the default pointing device cursor bitmap. If graphics tracking has been selected by using PDSetCursorType but no graphics cursor is loaded, the default cursor bitmap loaded by this operation is used.

For details on Mouse Services, see the *CTOS Programming Guide*.

Procedural Interface

PDLoadSystemCursor (pCursorShape,sCursorShape, bType): ercType

where

pCursorShape

is the memory address for an icon structure that defines the cursor shape.

sCursorShape

is the size of the structure that defines the cursor bitmap.

bType

is the type of cursor. The values of *bType* are

Value	**Description**
1	Mask Overlayed
2	Single Overlayed
3	XOR cursor

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntlInfo	1	6
1	rtInfo	1	0
2	nReqPbCb	1	1
3	nRespPbCb	1	0
4	userNum	2	
6	exchRet	2	
8	ercRet	2	
10	rqCode	2	32950
12	bType	2	
14	reserved	4	
18	pCursorShape	4	
22	sCursorShape	2	

PDQueryControls (pStruct, sStruct): ercType

Description

PDQueryControls allows an application to query various pointing device controls.

For details on Mouse Services, see the *CTOS Programming Guide*.

Procedural Interface

PDQueryControls (pStruct, sStruct): ercType

where

pStruct
sStruct

describe the memory area for the control structure. The format of the control structure is shown below:

Offset	Field	Size (bytes)	Description
0	bXGear	1	Speed gear for X
1	bYGear	1	Speed gear for Y

The speed gear is a value from 1 to 10, where 10 is the fastest speed for the mouse cursor.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntlInfo	1	6
1	rtInfo	1	0
2	nReqPbCb	1	1
3	nRespPbCb	1	0
4	userNum	2	
6	exchRet	2	
8	ercRet	2	
10	rqCode	2	32951
12	reserved	6	
18	pStruct	4	
22	sStruct	2	

PDQuerySystemControls (pStruct, sStruct): ercType

Description

PDQuerySystemControls allows an application to query various pointing device, system-wide controls.

For details on Mouse Services, see the *CTOS Programming Guide*.

Procedural Interface

PDQueryControls (pStruct, sStruct): ercType

where

pStruct
sStruct

describe the memory area for the control structure. The format of the control structure is shown below:

Offset	**Field**	**Size (bytes)**	**Description**
0	bDefaultXGear	1	Default X speed gear
1	bDefaultYGear	1	Default Y speed gear

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntlInfo	1	6
1	rtInfo	1	0
2	nReqPbCb	1	1
3	nRespPbCb	1	0
4	userNum	2	
6	exchRet	2	
8	ercRet	2	
10	rqCode	2	32952
12	reserved	6	
18	pStruct	4	
22	sStruct	2	

PDReadCurrentCursor (pCursorShape, sCursorShape, pbType): ercType

Description

PDReadCurrentCursor copies the current cursor bitmap for the application to the address specified. Status code 4809 is returned if sCursorShape is not large enough to hold the cursor.

For details on Mouse Services, see the *CTOS Programming Guide.*

Procedural Interface

PDReadCurrentCursor (pCursorShape, sCursorShape, pbType): ercType

where

pCursorShape
sCursorShape

describe the memory area for an icon structure that defines the cursor shape. The format of the icon structure is shown below:

Offset	Field	Size (bytes)	Description
0	wSignature	2	IC'
2	wVersion	2	0
4	sIcon	2	Size of icon bitmap
6	cbLine	2	Number of bytes per line in icon
8	bWidth	1	Icon width in pixels
9	bHeight	1	Icon height in pixels

Offset	Field	Size (bytes)	Description
10	bxOffset	1	Icon hot spot x offset from left
11	byOffset	1	Icon hot spot y offset from top
12	bMaskFlag	1	If bMaskFlag is the value 1, it specifies that the icon has a mask*
13	rgbReserved	3	
16	sbName	16	Name of icon
32	rgwBitmap	sIcon	Icon bitmap

* = not implemented

pbType

is the memory address of the cursor type. The values of *bType* are

1	Mask overlaid (AND/OR)
2	Single overlaid (OR)
3	Exclusive OR (XOR)

Request Block

sbType is always 2.

Offset	Field	Size (Bytes)	Contents
0	sCntlInfo	1	6
1	rtInfo	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	2
4	UserNum	2	
6	exchRet	2	
8	ercRet	2	
10	rqCode	2	32926
12	reserved	6	
18	pCursorShape	4	
22	sCursorShape	2	
24	pbType	4	
28	sbType	2	2

This page intentionally left blank

PdReadIconFile (fh, pIcon, sIcon, pWorkArea, sWorkArea, pcbRet): ercType

Description

PdReadIconFile reads the graphics cursor icon file. Status code 4809 is returned if *sIcon* is not large enough to hold the icon.

For details on Mouse Services, see the *CTOS Programming Guide*.

Procedural Interface

PdReadIconFile (fh, pIcon, sIcon, pWorkArea, sWorkArea, pcbRet): ercType

where

fh

is a file handle (word) returned from an OpenFile operation.

pIcon
sIcon

describe the memory area to which the icon structure is written. (For the format of the icon structure, see the description of PdLoadCursor.)

pWorkArea
sWorkArea

describe a 512 byte work area that is word aligned.

pcbRet

is the memory address where the count of bytes in the icon structure is returned.

Request Block

PDReadIconFile is an object module procedure in Mouse.lib.

PDSetVirtualCoordinates (wXMin, wYMin, wXMax, wYMax): ercType

Description

PDSetCharMapVirtualCoordinates defines a virtual coordinate space over the current character map window normalized coordinate space. This operation differs from PDSetVirtualCoordinates in that it compensates for systems that have the character map offset from the graphics bitmap (for example GC-003 systems). If you want to use the mouse in a character map application, use this operation instead of PDSetVirtualCoordinates.

For details on Mouse Services, see the *CTOS Programming Guide*.

Procedural Interface

PDSetVirtualCoordinates (wXMin, wYMin, wXMax, wYMax): ercType

where

wXMin
wYMin

specify the minimum value for the virtual screen coordinates.

wXMax
wYMax

specify the maximum value for the virtual screen coordinates.

Request Block

PDSetCharMapVirtualCoordinates is an object module procedure in Mouse.lib.

This page intentionally left blank

PDSetControls (pStruct, sStruct): ercType

Description

PDSetControls allows an application to set various pointing device controls.

For details on Mouse Services, see the *CTOS Programming Guide*.

Procedural Interface

PDSetControls (pStruct, sStruct): ercType

where

pStruct

describe the memory area for the control structure. The format of the control structure is shown below:

Offset	Field	Size (bytes)	Description
0	bXGear	1	Gear for X
1	bYGear	1	Gear for Y

Values for *bXGear* and *bYGear* set the speed at which the cursor tracks the motion of the mouse. Values range from 1 to 10, with 1 as the slowest gear. When the value of *bXGear* or *bYGear* equals 0, the gear is set to the system-wide default gear, which is set with the PDSetSystemControls operation. (See the description of PDSetSystemControls.)

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntlInfo	1	6
1	rtInfo	1	0
2	nReqPbCb	1	1
3	nRespPbCb	1	0
4	UserNum	2	
6	exchRet	2	
8	ercRet	2	
10	rqCode	2	32927
12	reserved	6	
18	pStruct	4	
22	sStruct	2	

PDSetCursorDisplay (fOn): ercType

Description

PDSetCursorDisplay turns the displayed cursor on and off. Tracking is not affected.

For details on Mouse Services, see the *CTOS Programming Guide*.

Procedural Interface

PDSetCursorDisplay (fOn): ercType

where

fOn

is a flag that is TRUE if the cursor is to be turned on. Otherwise, it is FALSE.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntlInfo	1	2
1	rtInfo	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	UserNum	2	
6	exchRet	2	
8	ercRet	2	
10	rqCode	2	32796
12	fOn	2	

This page intentionally left blank

PDSetCursorPos (wXPos, wYPos): ercType

Description

PDSetCursorPos sets the cursor position. Cursor coordinates are specified in virtual screen coordinates. Status code 4801 is returned if the cursor position is not in the window or on the screen. If cursor tracking is enabled, tracking continues from this new cursor position.

See also PDSetCursorPosNSC, which uses normalized screen coordinates.

For details on Mouse Services, see the *CTOS Programming Guide.*

Procedural Interface

PDSetCursorPos (wXPos, wYPos): ercType

where

wXPos
wYPos

are the cursor coordinates (words), in virtual screen coordinates.

Request Block

PDSetCursorPos is an object module procedure in Mouse.lib.

This page intentionally left blank

PDSetCursorPosNSC (wxPos, wYPos): ercType

Description

PDSetCursorPosNSC sets the cursor position. The cursor coordinates are specified in normalized screen coordinates. Status code 4801 is returned if the cursor position is not in the window or on the screen. If cursor tracking is enabled, tracking continues from this new cursor position.

See also PDSetCursorPos, which uses virtual screen coordinates.

For details on Mouse Services, see the *CTOS Programming Guide*.

Procedural Interface

PDSetCursorPosNSC (wxPos, wYPos): ercType

where

wXPos
wYPos

are the cursor coordinates (words), in virtual screen coordinates.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntlInfo	1	6
1	rtInfo	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	UserNum	2	
6	exchRet	2	
8	ercRet	2	
10	rqCode	2	32792
12	reserved	2	
14	wXPos	2	
16	wYPos	2	

PDSetCursorType (bType): ercType

Description

PDSetCursorType indicates the cursor type (character or graphics). The character cursor is a reverse video block, which cannot be changed. It shifts to half-brightness depending on the attribute of the character it is moving over so that it is always visible. The graphics cursor is an iconic cursor that can be defined by the programmer. The default shape is a left, upward-pointing arrow.

For details on Mouse Services, see the *CTOS Programming Guide*.

Procedural Interface

PDSetCursorType (bType): ercType

where

bType

specifies the cursor type. The values of *bType* are

Value	**Description**
0	Character cursor
1	Graphics cursor

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntlInfo	1	2
1	rtInfo	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	UserNum	2	
6	exchRet	2	
8	ercRet	2	
10	rqCode	2	32924
12	bType	2	

PDSetMotionRectangle (wXMin, wYMin, wDX, wDY): ercType

Description

PDSetMotionRectangle defines a motion rectangle in virtual screen coordinates. Status code 4801 is returned if the motion rectangle is not visible on the screen. When the cursor moves outside of a set motion rectangle, an event is returned by ReadInputEvent. (See ReadInputEvent.) The state is then returned to no motion rectangle.

See also PDSetMotionRectangleNSC, which uses normalized screen coordinates.

For details on Mouse Services, see the *CTOS Programming Guide*.

Procedural Interface

PDSetMotionRectangle (wXMin, wYMin, wDX, wDY): ercType

where

wXMin
wYMin

specify the minimum values (words) of the motion rectangle, in virtual screen coordinates.

wDX
wDY

specify the width (word) and height (word) of the motion rectangle, in virtual screen coordinates. The default setting for *wDX* and *wDY* is 0. If the value of either *wDX* or *wDY* equals 0, no motion rectangle is set. Consequently, no motion rectangle event is returned by the ReadInputEvent operation.

Request Block

PDSetMotionRectangle is an object module procedure in Mouse.lib.

PDSetMotionRectangleNSC (wXMin, wYMin, wDX, wDY): ercType

Description

PDSetMotionRectangleNSC defines a motion rectangle in normalized screen coordinates. When the cursor moves outside of a set motion rectangle, an event is returned by the ReadInputEvent operation. (See ReadInputEvent.) The state is then returned to no motion rectangle. Status code 4801 is returned if the motion rectangle is not visible on the screen.

See also PDSetMotionRectangle, which uses virtual screen coordinates.

For details on Mouse Services, see the *CTOS Programming Guide*.

Procedural Interface

PDSetMotionRectangleNSC (wXMin, wYMin, wDX, wDY): ercType

where

wXMin
wYMin

specify the minimum values (words) of the motion rectangle, in normalized screen coordinates.

wDX
wDY

specify the width (word) and height (word) of the motion rectangle, in normalized screen coordinates. If the value of either *wDX* or *wDY* equals 0, no motion rectangle is set. Consequently, no motion rectangle event is returned by ReadInputEvent.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntlInfo	1	8
1	rtInfo	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	UserNum	2	
6	exchRet	2	
8	ercRet	2	
10	rqCode	2	32793
12	wXMin	2	
14	wYMin	2	
16	wDX	2	
18	wDY	2	

PDSetSystemControls (pStruct, sStruct): ercType

Description

PDSetControls allows an application to set various system-wide pointing device controls.

See also PDSetControls, which sets nondefault gears. For details on Mouse Services, see the *CTOS Programming Guide*.

Procedural Interface

PDSetSystemControls (pStruct, sStruct): ercType

where

pStruct
sStruct

describe the memory area for the control structure. The format of the control structure is shown below:

Offset	Field	Size (bytes)	Description
0	bDefaultXGear	1	Default X gear
1	bDefaultYGear	1	Default Y gear

Values for *bDefaultXGear* and *bDefaultYGear* set the speed with which the cursor tracks the motion of the mouse. Values range from 1 to 10, with 1 as the slowest gear.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntlInfo	1	6
1	rtInfo	1	0
2	nReqPbCb	1	1
3	nRespPbCb	1	0
4	UserNum	2	
6	exchRet	2	
8	ercRet	2	
10	rqCode	2	32928
12	reserved	6	
18	pStruct	4	
22	sStruct	2	

PDSetTracking (fTracking): ercType

Description

PDSetTracking turns cursor tracking on and off. When tracking is turned off, the cursor continues to be displayed (unless the display has been disabled), but the cursor does not move with the mouse. The application program can still receive mouse cursor movement and events, and can call PDSetCursorPos.

(See the example in "Mouse Services" in the *CTOS Programming Guide*.)

Procedural Interface

PDSetTracking (fTracking): ercType

where

fTracking

is TRUE if the cursor is to be tracked. Otherwise, it is FALSE.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntlInfo	1	2
1	rtInfo	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	UserNum	2	
6	exchRet	2	
8	ercRet	2	
10	rqCode	2	32794
12	fTracking	2	

PDSetVirtualCoordinates (wXMin, wYMin, wXMax, wYMax): ercType

Description

PDSetVirtualCoordinates defines a virtual coordinate space over the normalized coordinate space. Values returned by the ReadInputEvent operation, for example, are in this range. (See ReadInputEvent.)

For details on Mouse Services, see the *CTOS Programming Guide*.

Procedural Interface

PDSetVirtualCoordinates (wXMin, wYMin, wXMax, wYMax): ercType

where

wXMin
wYMin

specify the minimum values (words) for the virtual screen coordinates.

wXMax
wYMax

specify the maximum values (words) for the virtual screen coordinates.

Request Block

PDSetVirtualCoordinates is an object module procedure in Mouse.lib.

This page intentionally left blank

PDTranslateNSCToVC (wNSCX, wNSCY, pwVCX, pwVCY): ercType

Description

PDTranslateNSCToVC translates normalized screen coordinates to virtual screen coordinates.

See also PDTranslateVCToNSC, which translates virtual screen coordinates to normalized screen coordinates. For details on Mouse Services, see the *CTOS Programming Guide*.

Procedural Interface

PDTranslateNSCToVC (wNSCX, wNSCY, pwVCX, pwVCY): ercType

where

wNSCX
wNSCY

specify the normalized screen coordinates (words).

pwVCX
pwVCY

specify the memory addresses to receive the translated virtual screen coordinates.

Request Block

PDTranslateNSCToVC is an object module procedure in Mouse.lib.

This page intentionally left blank

PDTranslateVCToNSC (wVCX, wVCY, pwNSCX, pwNSCY): ercType

Description

PDTranslateVCToNSC translates virtual screen coordinates to normalized screen coordinates.

See alsoPDTranslateNSCtoVC, which translates normalized screen coordinates to virtual screen coordinates. For details on Mouse Services, see the *CTOS Programming Guide*.

Procedural Interface

PDTranslateVCToNSC (wVCX, wVCY, pwNSCX, pwNSCY): ercType

where

wVCX
wVCY

specify the virtual screen coordinates (words).

pwNSCX
pwNSCY

specify the memory addresses to receive the translated normalized screen coordinates.

Request Block

PDTranslateVCToNSC is an object module procedure in Mouse.lib.

This page intentionally left blank

PosFrameCursor (iFrame, iCol, iLine): ercType

Description

PosFrameCursor establishes a visible cursor within the specified frame at the specified coordinates.

In a workstation, PosFrameCursor erases any previously displayed cursor, even one in another frame.

Procedural Interface

PosFrameCursor (iFrame, iCol, iLine): ercType

where

iFrame

specifies the frame. If *iFrame* = 0FFh, the coordinates *iCol* and *iLine* refer to virtual screen coordinates. This works exactly the same as if the caller has a frame that overlays the entire screen.

iCol
iLine

specify the horizontal and vertical coordinates within *iFrame* at which to establish a cursor. To remove the cursor from a frame, both *iCol* and *iLine* must be specified as 255 (TRUE).

Request Block

PosFrameCursor is a system–common procedure.

This page intentionally left blank

PostKbdTable (mode, pDataBlock, sDataBlock): ercType

Description

PostKbdTable directs the operating system to use an application-supplied translation or emulation data block while the application is running. Optionally, an application can again call PostKbdTable and specify that the default operating system data block be reused.

PostKbdTable is often used in conjunction with the ReadOsKbdTable operation. Typically, an application uses ReadOsKbdTable to obtain and modify a copy of the translation or emulation data block. By calling PostKbdTable, the application can then direct the operating system to use the modified data block. (See ReadOsKbdTable.)

PostKbdTable can also direct the operating system to use a data block that the end user has constructed with the Keyboard Customizing Tool.

If an emulation data block is in use, it must have the same ID number as the keyboard. The target ID in the emulation data block must match the keyboard ID in the translation data block When no emulation data block is in use, the ID of the translation data block must match the ID of the attached keyboard.

Status code 622 ("Posted ID invalid") is returned if the ID of the posted data block does not match the ID of the other data block currently in use. In this case, the operating system saves a pointer to the posted data block. However, the application can again call PostKbdTable to post a matching data block.

Status code 616 ("Bad keyboard ID") is returned if the emulation data block is not valid for the attached keyboard. When no emulation data block is in use, this status code is returned if the translation data block ID does not match the ID of the attached keyboard.

Status code 617 ("Bad keyboard data block") is returned if the data block of the application is damaged.

While a data block is in use by the operating system, a program must not deallocate the memory where the data block is stored. Fields within the data block may be changed by the operating system at any time.

Procedural Interface

PostKbdTable (mode, pDataBlock, sDataBlock): ercType

where

mode

specifies the type of data block to be posted. Each value indicates the following:

Value	Description
0	The data block is a translation data block.
1	The data block is an emulation data block.
2	This value directs the operating system to post a null emulation data block. This emulation data block is not used to generate emulated keyboard codes.

pDataBlock

is the memory address of the data block to be used. If *pDataBlock* is 0, PostKbdTable directs the operating system to use the default operating system data blocks.

sDataBlock

is the size of the keyboard data block to be used.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	1
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	400
12	mode	2	
14	reserved	4	
18	pDataBlock	4	
22	sDataBlock	2	

This page intentionally left blank

PrintAltMsg (pMwa, sMwa, iMsg, pRgSd, sRgSd, pBswa): ercType

Description

PrintAltMsg retrieves a message from the message file opened by calling the InitAltMsgFile operation and places the expanded message in a user-supplied byte stream. If the message contains macros, the macros are expanded with the corresponding string descriptor. (See Appendix F for a description of the message file macros.)

PrintAltMsg is similar to PrintMsg except that, with alternate message files, any number of message files may be open at any time. Access to each message file is determined by the Message Work Area (MWA).

Procedural Interface

PrintAltMsg (pMwa, sMwa, iMsg, pRgSd, sRgSd, pBswa): ercType

where

pMwa

is the memory address of the same MWA that was supplied to InitAltMsgFile.

sMwa

is the size (word) of the MWA. The size must be at least 60 bytes.

iMsg

is the message number (word).

pRgSd
sRgSd

describe an array of string descriptors. Each array element describes a string that is inserted into the message in place of a macro. (See Appendix F for details.)

String descriptors are 6-byte memory blocks. The first 4 bytes contain the memory address of the string, and the last 2 bytes contain string length.

pBswa

is the memory address of a Byte Stream Work Area of the open byte stream where the expanded message is written.

Request Block

PrintAltMsgFile is an object module procedure.

PrintErc (erc, pBSWA): ercType

Description

PrintErc retrieves an error message from a message file and places the expanded message in a user supplied byte stream. If the message contains macros, the macros are expanded with the corresponding string descriptor. (See Appendix F for a description of the message file macros.)

PrintErc is similar to PrintMsg except that, with PrintErc, the messages are retrieved from the error message file [Sys]<Sys>ErcMsg.bin (previously opened by calling InitErcFile) rather from the application's normal message file. This allows access to an error file without disturbing the application's message file.

If the error message file has not been previously opened, PrintErc will attempt to open the file and allocate the minimum memory needed to access it. If the application has previously allocated all available partition memory, the application must call InitErcMsg before calling GetErc.

Procedural Interface

PrintErc (erc, pBSWA): ercType

where

erc

is the status code (word).

pBSWA

is the memory address of a Byte Stream Work Area of the open byte stream where the expanded message is written.

Request Block

PrintErc is an object module procedure.

This page intentionally left blank

PrintFileClose: ercType

Description

PrintFileClose closes the output byte stream opened by the PrintFileOpen operation. (For details on use of the Print File operations, see "Utility Operations" in the *CTOS Operating System Concepts Manual*.)

Procedural Interface

PrintFileClose: ercType

Request Block

PrintFileClose is an object module procedure.

This page intentionally left blank

PrintFileOpen (pSdFileName): ercType

Description

PrintFileOpen opens a specified output byte stream in append mode so that output can be directed to this byte stream as well as to the video byte stream. This operation must be called before calling PrintFileStatus to set or query the status of the output byte stream. (For details on use of the Print File operations, see "Utility Operations" in the *CTOS Operating System Concepts Manual*.)

Procedural Interface

PrintFileOpen (pSdFileName): ercType

where

pSdFileName

is the memory address of an sd type structure describing the file name. The first 4 bytes of the structure contain the memory address of the file specification in the format {Node}[Vol]<Dir>FileName. The last two bytes contain the size of the specification.

Request Block

PrintFileOpen is an object module procedure.

This page intentionally left blank

PrintFileStatus (bCode, pfStatus): ercType

Description

PrintFileStatus returns or sets the current status of the byte stream previously opened with the PrintFileOpen operation. (For details on the use of the Print File operations, see "Utility Operations" in the *CTOS Operating System Concepts Manual.*)

Procedural Interface

PrintFileStatus (bCode, pfStatus): ercType

where

bCode

is a byte value that allows the caller to either query or update the status of the byte stream. The allowed values are

Value	**Description**
0	Allows the caller to query the status of the byte stream.
1	Allows the caller to change the status of the byte stream.

pfStatus

is a pointer to a status byte that has one of the following meanings depending on the value of *bCode*:

Value of *bCode*	**Meaning of** *pfStatus*
0	Set the flag pointed to by *pfStatus* as follows: TRUE means to direct output to the video and a non-video bytestream. FALSE means to direct output to the video only.
1	Change the status of the output bytestream as follows: TRUE means means to resume directing output to the non-video bytestream. The user must have previously called PrintFileOpen. FALSE means to cease directing output to the non-video bytestream.

Request Block

PrintFileStatus is an object module procedure.

PrintMsg (iMsg, pRgSd, sRgSd, pBswa): ercType

Description

PrintMsg retrieves a message from the message file and places the expanded message in a user supplied byte stream. If the message contains macros, that is number(s) and/or letter(s) preceded by a percent sign (%), the macros are expanded with the corresponding string descriptor. (See Appendix F for a description of the message file macros.)

PrintMsg is similar to GetMsg, except that GetMsg places the expanded message in memory. (See GetMsg.)

Procedural Interface

PrintMsg (iMsg, pRgSd, sRgSd, pBswa): ercType

where

iMsg

is the message number.

pRgSd
sRgSd

describe an array of string descriptors. Each element of the array describes a string that is inserted into the message in place of a macro. (See Appendix F.)

String descriptors are 6-byte blocks of memory in which the first 4 bytes contain the memory address of the string and the last 2 bytes contain the size of the string.

pBswa

is the memory address of a Byte Stream Work Area of the open byte stream where the expanded message is written.

Request Block

PrintMsg is an object module procedure.

ProgramColorMapper (pNewPalette, sNewPalette, pNewControl, sNewControl, pOldPaletteRet, sOldPaletteRet, pOldControlRet, sOldControlRet): ercType

Caution: *This operation works on workstation hardware only. Do not use it on shared resource processor hardware.*

Description

ProgramColorMapper sets and queries the color palette and/or color control structure. Through the ProgramColorMapper interface, the caller can enable alphanumeric and/or graphics displays and can define customized colors in the color palette(s) for the displays. The caller can choose to set up color using either a single-palette or three-palette format. The single palette format allows selection from a greater range of colors and color intensities on hardware that supports a large number of colors and intensities.

(The parameter descriptions below assume that you are familiar with the ProgramColorMapper structures. For details on the formats of the palette and control structures and examples of how to use ProgramColorMapper, see "Using Color" in the *CTOS Programming Guide.*)

Using this operation causes ASCB.fExecScreen to be set to 0.

Procedural Interface

ProgramColorMapper (pNewPalette, sNewPalette, pNewControl, sNewControl, pOldPaletteRet, sOldPaletteRet, pOldControlRet, sOldControlRet): ercType

where

pNewPalette
sNewPalette

describe the new palette.

If the new control structure described by *pNewControl* and *sNewControl* defines the three-palette format (the control structure field *bFormat* is 0 or *sNewControl* is less than 6), *sNewPalette* may have one of the following sizes:

Size	**Meaning**
0	No palette change.
8	Replace the alpha palette.
16	Replace the alpha and graphics1 palettes.
24	Replace all three palettes.

If the new control structure defines the single-palette format (the control structure field *bFormat* is 1 and *sNewControl* is greater than 5), *sNewPalette* may have one of the following sizes:

Size	**Meaning**
0	No palette change.
3*x	Replace x of the 3-byte entries in the palette starting at the palette entry specified by the value of *wIndexStart* in the control structure.

pNewControl
sNewControl

describe the new control structure. *sNewControl* may be one of the following values:

Size	Meaning
0	No change
5	Three-palette format
8	Single-palette format if the field *bFormat* is 1

pOldPaletteRet
sOldPaletteRet

describe the memory area where a copy of the current palette will be returned. If *sOldPaletteRet* is 0, no current palette is returned. The operation works correctly even if *pOldPaletteRet* is the same as *pNewPalette*.

To query about old palette in single-palette format, the user needs to inform ProgramColorMapper via *pNewControl* and *sNewControl* (i.e., *sNewControl* should be set to 8 and *NewControl.bformat* should be set to 1). For example

> ProgramColorMapper (0, 0, *pNewControl*, 8, *pOldPalette*, *sOldPalette*, *pOldControl*, 8)

This queries *pOldPalette* in single-palette format. In this example, *OldControl.bformat* = 1. To query the old palette information in three-palette format, the following is sufficient

> ProgramColorMapper (0, 0, 0, 0, *pOldPalette*, *sOldPalette*, *pOldControl*, 5)

where *OldControl.bformat* = 0.

pOldControlRet
sOldControlRet

describe the memory area where a copy of the current control structure will be returned. If *sOldControlRet* is 0, no control information is returned.

Request Block

ProgramColorMapper is an object module procedure.

PSCloseSession (sh): ercType

Description

PSCloseSession closes the statistics gathering session with the Performance Statistics system service. The session was previously opened by a call to the PSOpenStatSession or the PSOpenLogSession operation.

For details on how to use the Performance Statistics system service, see the *CTOS Programming Guide, Volume 2.*

Procedural Interface

PSCloseSession (sh): ercType

where

sh

is the session handle returned by PSOpenStatSession or PSOpenLogSession.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	2	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	8197
12	sh	2	

PSDeinstServer (pPSUserNumRet): ercType

Description

PSDeinstServer deinstalls the Performance Statistics system service.

For details on how to use the Performance Statistics system service, see the *CTOS Programming Guide, Volume 2.*

Procedural Interface

PSDeinstServer (pPSUserNumRet): ercType

where

pPsUserNumRet

is the memory address to which the user number of the Performance Statistics system service is returned.

Request Block

sPsUserNumRet is always 2.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	8198
12	reserved	6	
18	pPSUserNumRet	4	
22	sPSUserNumRet	2	2

PSend (exchange, pMsg): ercType

Description

The PSend primitive is identical to Send but should be used instead of Send for interrupt handling.

PSend checks whether processes are queued at the specified exchange. If processes are queued, then the process that was queued first is removed from the queue, given the memory address of the message, and placed into the ready state. If such a process has a higher priority than the calling process, it is scheduled for immediate execution and the calling process remains preempted until the higher priority process reenters the waiting state.

If no processes are waiting at the exchange, then the message is queued at the exchange.

Procedural Interface

PSend (exchange, pMsg): ercType

where

exchange

is the identification of the exchange where the message is sent.

pMsg

is the memory address of the message (or a 4-byte field of information whose interpretation is agreed upon by the sending and receiving processes).

Request Block

PSend is a Kernel primitive.

PSGetCounters (sh, pbBlockId, cbBlockId, pbCountRet, cbCountRet): ercType

Description

PSGetCounters returns the memory address of the counters for the specified blocks and indexes. Each counter is a double word. The block-index combinations must have been specified previously in the call to the PSOpenStatSession operation, which opened the statistics gathering session. Note that PSGetCounters only requires that the caller specify block numbers and indexes. This information is a subset of that specified in the call to PSOpenStatSession.

For details on how to use the Performance Statistics system service, see the *CTOS Programming Guide, Volume 2.* The format of the Performance Statistics structure is shown in Chapter 4, "System Structures," in this manual.

Procedural Interface

PSGetCounters (sh, pbBlockId, cbBlockId, pbCountRet, cbCountRet): ercType

where

sh

is the session handle returned by PSOpenStatSession.

pbBlockId
cbBlockId

describe an array of block-indexes for which information is requested. The elements in this array must identify blocks already described in the call to PSOpenStatSession. (See the arguments *pbBlockDesc/cbBlockDesc* in the description of PSOpenStatSession.) The format of a block ID array element is shown below:

Offset	**Field**	**Size (bytes)**	**Description**
0	blockNum	2	Is the number of a block specified in PSOpenStatSession.
2	index	2	Is the index of the block specified by *blockNum*.

pbCountRet
cbCountRet

describe the memory area where the counters are to be returned. The counters are returned in the order in which the block IDs are given in the array described by *pbBlockId/cbBlockId*. That is, statistics for the block described by the first array element are returned first, followed by statistics for the second array element, and so forth. *cbCountRet* must be the sum of the byte counts of the block-indexs as specified in the call to PSOpenStatSession.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	2	6
1	RtCode	1	0
2	nReqPbCb	1	1
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	8195
12	sh	2	
14	reserved	4	
18	pbBlockId	4	
22	cbBlockId	2	
24	pbCountRet	4	
28	cbCountRet	2	

This page intentionally left blank

PSOpenLogSession (pbDevName, cbDevName, wBlockID, wIterations, wLogHeapSize, pbShRet): ercType

Description

PSOpenLogSession opens a new logging session for the caller. The session can log either of the following:

- the currently active processes in the ready-to-run queue (logged at 1 millisecond intervals)
- the memory usage of short-lived and long-lived memory (logged at 1/10th second intervals)

The caller specifies the number of iterations and the number of bytes (in 512 byte multiples) needed to store the logged information.

Procedural Interface

PSOpenLogSession (pbDevName, cbDevName, wBlockID, wIterations, wLogHeapSize, pbShRet): ercType

where

pbDevName
cbDevName

describe the node and volume for which the information is requested in the format {NodeName}[DevName].

wBlockId

specifies the type of log to be obtained. *wBlockId* is one of the following values:

Value	**Description**
10	active processes in the run queue
11	memory usage

wIterations

specifies the number of iterations the caller is interested in logging.

wLogHeapSize

is the size of the log heap in 512 byte multiples.

pbShRet

describes the memory area where the session handle for the opened logging session is returned.

Request Block

cbShRet is always 2.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	1
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	8199
12	wBlockId	2	
14	wIterations	2	
16	wHeapSize	2	
18	pbDevName	4	
22	cbDevName	2	
24	pbShRet	4	
28	cbShRet	2	2

This page intentionally left blank

PSOpenStatSession (pbDevName, cbDevName, pbBlockDesc, cbBlockDesc, pbShRet): ercType

Description

PSOpenStatSession opens a session for which performance statistics are collected by the Performance Statistics system service. The statistics include measurements of I/O (disk, device, and file system) and processing (processes, programs, and partitions). The session handle returned is used in subsequent calls to the Performance Statistics system service.

For details on how to use the Performance Statistics system service, see the *CTOS Programming Guide, Volume 2.*

Procedural Interface

PSOpenStatSession (pbDevName, cbDevName, pbBlockDesc, cbBlockDesc, pbShRet): ercType

where

pbDevName
cbDevName

describe the node and volume for which the information is requested in the format {NodeName}[DevName].

pbBlockDesc
cbBlockDesc

describe an array containing a description of the Performance Statistics structure blocks requested. (For the format of the Performance Statistics structure, see in Chapter 4, "System Structures.")

Each array element specifies information about a given block. An element specifies the block number, the index, the offset from the beginning of that block of the information needed, and the size (in 4-byte multiples) of information needed. The format of an array element is shown below:

Offset	Field	Size (bytes)	Description
0	blockNum	2	Is the number of the block for which information is requested. For example, the caller would specify the value 2 (for block number 2) to obtain statistics on disk errors and I/O blocks.
2	index	2	Is the index. The index is always 0 for block number 1, which returns CPU statistics. The index for blocks 2 through 9 is the number of the device (0 through 17) relative to the CPU for which statistics are being requested.
4	offset	2	Is the offset from the beginning of this block of the information needed. For each block, offsets start at 0, and there is an offset every 4 bytes. At each offset, a new type of statistics is given. For example, block number 2, device index 0, starting at offset 0 for 4 bytes gives the number of hard disk errors.
6	byteCount	2	Indicates the amount of statistics requested. For example, block number 2, device index 0, starting at offset 0 for 8 bytes gives the number of hard disk errors and the number of soft disk errors.

pbShRet

is the memory address to which the session handle for the opened statistics session is to be returned.

Request Block

cbShRet is always 2.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	2
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	8194
12	reserved	6	
18	pbDevName	4	
22	cbDevName	2	
24	pbBlockDesc	4	
28	cbBlockDesc	2	
30	pbShRet	4	
34	cbShRet	2	2

This page intentionally left blank

PSReadLog (sh, pbLogData, cbLogData, pbDataCntRet): ercType

Description

PSReadLog obtains the log for the process run queue or memory usage for the logging session previously opened by a call to PSOpenLogSession.

Procedural Interface

PSReadLog (sh, pbLogData, cbLogData, pbDataCntRet): ercType

where

sh

is the session handle returned by PSOpenLogSession.

pbLogData
cbLogData

describe the memory area where the information is to be returned. The structure of the data returned shown below is for process activity and for memory usage. *cbLogData* should be a multiple of 512.

The following are the data returned on process activity (caller specified 10 for the value of *wBlockId* in the call to PSOpenLogSession):

Offset	Field	Size (bytes)	Description
0	BlockNum	2	Block number
2	iterationNum	2	Number of requested iterations
4	iterationSuccess	2	Number of successful iterations

Data on each successful iteration of process activity is returned in the following format:

Field	Size (bytes)	Description
processNum	2	Number of processes for this iteration. Data on each process is returned in the following format:

Field	Size (bytes)
User Number	2
Partition Name	13
Priority	1

The following are the data returned on memory usage (caller specified 11 for the value of *wBlockId* in the call to PSOpenLogSession):

Offset	Field	Size (bytes)	Description
0	BlockNum	2	Block number
2	iterationNum	2	Number of requested iterations
4	iterationSuccess	2	Number of successful iterations

Data on each successful iteration of memory usage is returned in the following format:

Field	Size (bytes)	Description
partitionNum	4	Number of partitions for this iteration. Data on each partition is returned in the following format:

Field	Size (bytes)
User Number	2
cbLongLived	4
cbShortLived	4

pbDataCntRet

is the memory address to which the actual length of the logged information read is returned.

Request Block

lfa is always 0, and *cbDataCntRet* is always 2.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	1
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	8200
12	sh	2	
14	lfa	4	0
18	pbLogData	4	
22	cbLogData	2	
24	pbDataCntRet	4	
28	cbDataCntRet	2	2

PSResetCounters (sh, pbBlockId, cbBlockId): ercType

Description

PSResetCounters resets the performance statistics counters for the specified block-index(s).

For details on how to use the Performance Statistics system service, see the *CTOS Programming Guide, Volume 2.* The format of the Performance Statistics structure is shown in Chapter 4, "System Structures," in this manual.

Procedural Interface

PSResetCounters (sh, pbBlockId, cbBlockId): ercType

where

sh

is the session handle returned by PSOpenStatSession.

pbBlockId
cbBlockId

describe an array of block-indexes for which statistics counters are to be reset. The elements in this array must identify blocks already described in the call to PSOpenStatSession. (See the arguments *pbBlockDesc/cbBlockDesc* in the description of PSOpenStatSession.) The structure of a block ID array element is shown below:

Offset	Field	Size (bytes)	Description
0	blockNum	2	Is the number of a block specified in PSOpenStatSession.
2	index	2	Is the index of the block specified by *blockNum*.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	1
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	8196
12	sh	2	
14	reserved	4	
18	pbBlockId	4	
22	cbBlockId	2	

PurgeMcr: ercType

Description

PurgeMcr flushes the internal magnetic card reader (MCR) buffer of the operating system. The buffer is flushed automatically by the operating system every 20 seconds and by the next swipe of a card through the reader. This operation allows an application to flush the buffer at any time.

Procedural Interface

PurgeMcr: ercType

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	2	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	8236*

*On operating systems prior to CTOS I 3.3, the request code is 65522 or −14.

This page intentionally left blank

PutBackByte (pBswa, b): ercType

Description

PutBackByte returns one byte to the open input byte stream identified by the memory address of the Byte Stream Work Area. This can be useful to a program such as a compiler that may decide, after looking at a character, that it should be processed by a different operation. Only one byte can be put back before reading again. An attempt to put back more than one byte returns status code 2305 ("Too many put backs").

Procedural Interface

PutBackByte (pBswa, b): ercType

where

pBswa

is the memory address of the same Byte Stream Work Area that was supplied to OpenByteStream.

b

is the 8-bit byte to be put back.

Request Block

PutBackByte is an object module procedure.

This page intentionally left blank

PutByte (b, base)

NOTE: This operation does not return a status code.

Description

PutByte prints a byte (8-bit unsigned integer) value to the video or other device as specified with the NPrint and PutChar operations.

Procedural Interface

PutByte (b, base)

where

b

  is a byte.

base

  is the base (2 to 16) in which the byte value is to be displayed.

Request Block

PutByte is an object module procedure.

This page intentionally left blank

PutChar (ch)

NOTE: This operation does not return a status code.

Description

PutChar prints a character to the video or other device.

If a file was opened using the PrintFileOpen operation, PutChar writes a character to the file as well as to the video device.

(See also NPrint.)

Procedural Interface

PutChar (ch)

where

ch

is a (byte) character to be written.

Request Block

PutChar is an object module procedure.

This page intentionally left blank

PutCharsAndAttrs (iFrame, iCol, iLine, pbText, cbText, pbAttrs, cbAttrs): ercType

NOTES:

1. *This operation is similar to PutFrameCharsAndAttrs except that it is a system library procedure that works on all operating system versions. This operation is for use by programs that need to execute on real mode operating systems.*
2. *When using standard VGA on EISA/ISA-bus workstations, this operation may display some attributes (specifically, bold, underlining, struckthrough, and blinking) differently than expected. System configuration options allow you to substitute color for these attributes. See the* CTOS System Administration Guide *for more information.*

Description

PutCharsAndAttrs combines the functions of the PutFrameChars and PutFrameAttrs operations so that a sequence, such as a line of characters and attributes, can be written in a single call.

When PutCharsAndAttrs is called, it, in turn, invokes the operation PutFrameCharsAndAttrs. However, if PutFrameCharsAndAttrs is not supported by the operating system, the library code in the PutCharsAndAttrs procedure writes the character and attribute data. Because of this, video filters designed to work with PutFrameCharsAndAttrs will still work with programs that use PutCharsAndAttrs.

Procedural Interface

PutCharsAndAttrs (iFrame, iCol, iLine, pbText, cbText, pbAttrs, cbAttrs): ercType

where

iFrame

is a word that specifies either a frame or the virtual screen. If *iFrame* = 0FFh, the coordinates *iCol* and *iLine* refer to virtual screen coordinates. This works exactly the same as if the caller has a frame that overlays the entire screen.

iCol

specifies the horizontal coordinate (word) within *iFrame* where the first character of the text string is to be moved.

iLine

specifies the vertical coordinate (word) within *iFrame* where the first character of the text string is to be moved.

pbText
cbText

describe the text string to be moved into the character map.

pbAttrs
cbAttrs

describe a string that contains the attributes to be associated with the text string. If *cbAttrs* = 1, the programmer supplies a single attribute that applies to every character in the string. If *cbAttrs* = *cbText*, the programmer supplies a separate attribute for each character in the string.

Request Block

PutCharsAndAttrs is an object module procedure.

This page intentionally left blank.

PutFrameAttrs (iFrame, iCol, iLine, attr, nPos): ercType

NOTE: When using standard VGA on EISA/ISA-bus workstations, this operation may display some attributes (specifically, bold, underlining, struckthrough, and blinking) differently than expected. System configuration options allow you to substitute color for these attributes. See the System Administration Guide for more information.

Description

PutFrameAttrs establishes the same character attribute for a range of character positions within a specified frame. The character attribute is applied first left to right and then top to bottom in the same manner as characters are moved into a frame.

Procedural Interface

PutFrameAttrs (iFrame, iCol, iLine, attr, nPos): ercType

where

iFrame

specifies the frame. If *iFrame* = 0FFh, the coordinates *iCol* and *iLine* refer to virtual screen coordinates. This works exactly the same as if the user has a frame that overlays the entire screen.

iCol
iLine

specify the horizontal and vertical coordinates within *iFrame* at which to begin altering character attributes.

attr

the low-order 6 bits of attr specify the character attributes.

The interpretation of bits 0 through 3 is as follows:

Bit	Value	Attribute
0	1	Half-bright. (Note that if screen half-bright is set, the interpretation of the character attribute halfbright is to negate half-bright (that is, to display the character at full brightness.)
1	2	Underlining.
2	4	Reverse video. (Note that if screen reverse video is set, the interpretation of the character attribute reverse video is to negate reverse video (that is, to display a light character on a dark background.)
3	8	Blinking on character map workstations. This value means outlined on bit map workstations, such as the GC003.

For a workstation, the interpretation of bits 4 and 5 is as follows:

Bit	Value	Attribute
4	16	Bold
5	32	Struck-through

nPos

specifies the number of character positions whose character attributes are to be changed.

Request Block

PutFrameAttrs is a system-common procedure.

PutFrameChars (iFrame, iCol, iLine, pbText, cbText): ercType

NOTE: To draw a line or a box, an application should use EnlsDrawLine or EnlsDrawBox in place of PutFrameChars. EnlsDrawLine, EnlsDrawBox, and other ENLS operations allow an application to run in different language environments without changes to the program code.

Description

PutFrameChars overwrites the specified character positions in the specified frame with the specified text string. PutFrameChars does not cause the character attributes associated with the character positions to change and never causes scrolling.

Procedural Interface

PutFrameChars (iFrame, iCol, iLine, pbText, cbText): ercType

where

iFrame

specifies the frame. If *iFrame* = 0FFh, the coordinates *iCol* and *iLine* refer to virtual screen coordinates. This works exactly the same as if the caller has a frame that overlays the entire screen.

iCol
iLine

specify the horizontal and vertical coordinates within *iFrame* at which the first character of the text string is to be moved.

pbText
cbText

describe the text string to be moved into the character map.

Request Block

PutFrameChars is a system–common procedure.

PutFrameCharsAndAttrs (iFrame, iCol, iLine, pbText, cbText, pbAttrs, cbAttrs): ercType

NOTES:

1. *This operation is supported by protected mode operating systems only. Applications that need to run on real mode operating systems should use PutCharsAndAttrs.*
2. *When using standard VGA on EISA/ISA-bus workstations, this operation may display some attributes (specifically, bold, underlining, struckthrough, and blinking) differently than expected. System configuration options allow you to substitute color for these attributes. See the* CTOS System Administration Guide *for more information.*
3. *To draw a string of form characters, an application should use EnlsDrawFormChars in place of PutFrameCharsAndAttrs. EnlsDrawFormChars and other ENLS operations allow an application to run in different language environments without changes to the program code.*

Description

PutFrameCharsAndAttrs combines the functions of the PutFrameChars and PutFrameAttrs operations so that a sequence, such as a line of characters and attributes, can be written in a single call.

Procedural Interface

PutFrameCharsAndAttrs (iFrame, iCol, iLine, pbText, cbText, pbAttrs, cbAttrs): ercType

where

iFrame

specifies either a frame or the virtual screen. If *iFrame* = 0FFh, then the coordinates *iCol* and *iLine* refer to virtual screen coordinates. This works exactly the same as if the caller has a frame that overlays the entire screen.

iCol
iLine

specify the horizontal and vertical coordinates within *iFrame* where the first character of the text string is to be moved.

pbText
cbText

describe the text string to be moved into the character map.

pbAttrs
cbAttrs

describe a string that contains the attributes to be associated with the text string. If *cbAttrs* = 1, the programmer supplies a single attribute that applies to every character in the string. If *cbAttrs* = *cbText*, the programmer supplies a separate attribute for each character in the string.

Request Block

PutFrameCharsAndAttrs is a system-common procedure.

PutPointer (p, base)

NOTE: This operation does not return a status code.

Description

PutPointer prints a memory address to the video or other device as specified by the NPrint and PutChar operations.

Procedural Interface

PutPointer (p, base)

where

p

is a memory address.

base

is the base (2 to 16) in which the memory address is to be displayed.

Request Block

PutPointer is an object module procedure.

This page intentionally left blank

PutQuad (q, base)

NOTE: This operation does not return a status code.

Description

PutQuad prints a quad (32-bit unsigned integer) value to the video or other device as specified by the NPrint and PutChar operations.

Procedural Interface

PutQuad (q, base)

where

q

is a quad.

base

is the base (2 to 16) in which the quad value is to be displayed.

Request Block

PutQuad is an object module procedure.

This page intentionally left blank

PutWord (w, base)

NOTE: This operation does not return a status code.

Description

PutWord prints a word (16-bit unsigned integer) value to the video or other device as specified by the NPrint and PutChar operations.

Procedural Interface

PutWord (w, base)

where

w

is a word.

base

is the base (2 to 16) in which the word value is to be displayed.

Request Block

PutWord is an object module procedure.

This page intentionally left blank

QueryBigMemAvail (pqRet): ercType

Description

QueryBigMemAvail returns the size in bytes of all available free memory in an application partition. Size is a function of the following:

- physical memory size
- limits specified in the run file containing the QueryBigMemAvail operation
- limits specified at partition creation time

The following operations allocate free memory:

AllocAllMemorySL
AllocAreaSL
AllocMemoryLL
AllocMemorySL
ExpandMemoryLL.
ExpandAreaSL

Programs that operate in both real and protected modes should use this operation rather than QueryMemAvail because the latter operation can report a maximum size of only 1M-byte.

Procedural Interface

QueryBigMemAvail (pqRet): ercType

where

pqRet

is the memory address to which the four-byte amount of memory available is to be returned.

Request Block

sqMax is always 4.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	283
12	reserved	6	
18	pqRet	4	
22	sqMax	2	4

QueryBoardInfo (wSlot, wCase, pbInfoRet, cbInfoMax): ercType

NOTE: Programs that may run on earlier versions of the operating system should use the object module procedure GetBoardInfo rather than this request to obtain the same information.

Description

QueryBoardInfo accepts the slot number of an SRP processor and a case value. Based on the case value specified, information is returned about the specified processor in the memory area provided. If the memory area is too small to contain all the information returned by this operation, the information is truncated.

Procedural Interface

QueryBoardInfo (wSlot, wCase, pbInfoRet, cbInfoMax): ercType

where

wSlot

is the slot number (word) of the processor. (For an illustration of slot numbering on shared resource processors, see the *CTOS System Administration Guide*.)

wCase

is a word value with 0 being the only value currently supported. The structure returned by case 0 contains the following information for the processor specified by *wSlot*:

Offset	Field	Size (bytes)	Description
0	localUserNumMax	2	Is the maximum number of local user numbers.
2	iCpu	2	Is the CPU number. (See the description of *iCpu* in the structure returned by the QueryUserLocation operation.)
4	fMfProcessor	1	Is a flag that is TRUE if the processor is the master processor.

pbInfoRet
cbInfoMax

describe the memory area to which the structure specified by *wCase* is returned.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	rtCode	1	0
2	nReqPbCb	0	0
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	314
12	wSlot	2	
14	wCase	2	
16	reserved	2	
18	pbInfoRet	4	
22	cbInfoMax	2	

This page intentionally left blank

QueryBounds (iFrame, pcCols, pcLines): ercType

NOTE: This operation is similar to QueryFrameBounds except that it is a system library procedure that works on all operating system versions.

Description

QueryBounds returns the size (number of columns and lines) for the specified frame.

Procedural Interface

QueryBounds (iFrame, pcCols, pcLines): ercType

where

iFrame

specifies the frame (word) for which the size is returned. If *iFrame* = 0FFh, the size of the virtual screen is returned. This works exactly the same as if the caller has a frame that overlays the entire screen.

pcCols

is the memory address of the word where the number of columns is to be returned.

pcLines

is the memory address of the word where the number of lines is to be returned.

Request Block

QueryBounds is an object module procedure.

QueryCharsAndAttrs (iFrame, iCol, iLine, pbText, cbText, pbAttrs, cbAttrs): ercType

NOTE: This operation is similar to QueryFrameCharsAndAttrs except that it is a system library procedure that works on all operating system versions. This operation is for use by programs that need to execute on real mode operating systems.

Description

QueryCharsAndAttrs returns a character string and its associated attributes from the character map at the specified coordinates. QueryCharsAndAttrs complements the PutCharsAndAttrs operation.

Procedural Interface

QueryCharsAndAttrs (iFrame, iCol, iLine, pbText, cbText, pbAttrs, cbAttrs): ercType

where

iFrame

is a word that specifies either a frame or the virtual screen. If *iFrame* = 0FFh, the coordinates *iCol* and *iLine* refer to virtual screen coordinates. This works exactly the same as if the caller has a frame that overlays the entire screen.

If *iFrame* = 0FEh, the coordinates *iCol* and *iLine* refer to real screen coordinates. This option may be used to obtain a snapshot of the real screen. Note that the virtual screen is the screen that belongs to the user, and the real screen is the screen that is visible.

iCol

specifies the horizontal coordinate (word) at which the first character is to be returned.

iLine

specifies the vertical coordinate (word) at which the first character is to be returned.

pbText
cbText

describe the character buffer where the text string is to be returned.

pbAttrs
cbAttrs

describe the attribute data buffer where the attributes associated with the text string will be returned.

NOTE: If cbText <> 0 and cbAttrs <> 0, cbAttrs must equal cbText. Setting cbText equal to 0 allows the query of attributes only. Likewise if cbText <> 0 and cbAttrs = 0, this is a character query only.

Request Block

QueryCharsAndAttrs is an object module procedure.

QueryCoprocessor (pbStatusRet): ercType

Description

QueryCoprocessor returns the coprocessor status of the workstation.

Procedural Interface

QueryCoprocessor (pbStatusRet) : ercType

where

pbStatusRet

is the memory address of a 1–byte memory area where the coprocessor status is returned.

The returned coprocessor status values are:

0	–Coprocessor not present
1	–Unknown coprocessor
2	–80287 coprocessor present
3	–80387 coprocessor present
4 - 255	–Reserved

Request Block

QueryCoprocessor is a system–common procedure.

This page intentionally left blank

QueryCursor (iFrame, piCol, piLine): ercType

NOTE: This operation is similar to QueryFrameCursor except that it is a system library procedure that works on all operating system versions.

Description

QueryCursor returns the cursor position for the specified frame.

Procedural Interface

QueryCursor (iFrame, piCol, piLine): ercType

where

iFrame

specifies the frame for which the cursor position is returned. If *iFrame* is TRUE (255), the coordinates returned at *piCol* and *piLine* refer to virtual screen coordinates. This works exactly the same as if the caller has a frame that overlays the entire screen.

NOTE: If the value 0FFh is returned for both iCol *and* iLine, *the cursor is not present in the specified frame.*

piCol

is the memory address of a word at which the horizontal coordinate of the cursor within the frame will be returned.

piLine

is the memory address of a word at which the vertical coordinate of the cursor within the frame will be returned.

Request Block

QueryCursor is an object module procedure.

QueryDaLastRecord (pDawa, pqiRecordRet): ercType

Description

QueryDaLastRecord copies the number of the last record in the open DAM file to the specified area. The file is identified by the memory address of the Direct Access Work Area. The last record is the existing record having the largest record number.

If the DAM file contains no records, the last record number is 0.

Procedural Interface

QueryDaLastRecord (pDawa, pqiRecordRet): ercType

where

pDawa

is the memory address of the same Direct Access Work Area that was supplied to OpenDaFile.

pqiRecordRet

is the memory address of the 32-bit memory area where the last record number is written.

Request Block

QueryDaLastRecord is an object module procedure.

This page intentionally left blank

QueryDaRecordStatus (pDawa, qiRecord, pStatusRet): ercType

Description

QueryDaRecordStatus copies the status of a record in the open DAM file to the specified area. The file is identified by the memory address of the Direct Access Work Area (DAWA). The record status is interpreted in this way:

Status code	**Description**
0	"ercOK" (The record exists.)
3302	"ercRecordDoesNotExist" (The record does not exist.)
3007	"ercRecordBeyondExistingRecords" (the record does not exist. The record has a larger record number than any existing record.)

Caution: *The status code value returned by QueryDaRecordStatus is the status of the operation, not the record status. The memory address of the record status is passed as a parameter.*

Procedural Interface

QueryDaRecordStatus (pDawa, qiRecord, pStatusRet): ercType

where

pDawa

is the memory address of the same Direct Access Work Area that was supplied to OpenDaFile.

qiRecord

is a 32-bit unsigned integer specifying the number of the record to query.

pStatusRet

is the memory address of a word where the record status is written.

Request Block

QueryDaRecordStatus is an object module procedure.

QueryDcb (pbDevSpec, cbDevSpec, pDcbRet, sDcbMax): ercType

Caution: *This operation is for* **restricted use only** *and is subject to change in future operating system releases. New applications should use QueryDiskGeometry instead of QueryDcb.*

Description

QueryDcb copies the Device Control Block of the specified device to the specified memory area. If the specified area is not large enough to hold the requested information, the information is truncated.

QueryDcb does not require a password. To avoid security violations, 0's are returned in the *devPassword* field of the Device Control Block.

(See Chapter 4, "System Structures," for the format of the Device Control Block.)

Procedural Interface

QueryDcb (pbDevSpec, cbDevSpec, pDcbRet, sDcbMax): ercType

where

pbDevSpec
cbDevSpec

describe a character string of the form {Node}[DevName] or {Node}[VolName]. Square brackets are optional for the device name. The distinction between uppercase and lowercase is not significant in matching device names.

pDcbRet
sDcbMax

describe the memory area for the return of the Device Control Block.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	16	
1	RtCode	1	0
2	nReqPbCb	1	1
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	124
12	reserved	6	
18	pbDevSpec	4	
22	cbDevSpec	2	
24	pDcbRet	4	
28	sDcbMax	2	

QueryDefaultRespExch (pExchRet): ercType

Description

QueryDefaultRespExch allows a process to determine the identification of its own default response exchange.

Procedural Interface

QueryDefaultRespExch (pExchRet): ercType

where

pExchRet

is the memory address of a word where the identification of the default response exchange of the inquiring process is returned.

Request Block

QueryDefaultRespExch is a system–common procedure.

This page intentionally left blank

QueryDeviceName (pbRouteSpec, cbRouteSpec, iDevice, pbNamesRet, cbNamesMax, pcbNamesRet): ercType

Description

QueryDeviceName returns the name entry in a workstation or shared resource processor CPU device name table for each device specified. The format of each name entry returned is as follows:

Offset	**Field**	**Size (Bytes)**	**Contents**
0	CPU index	1	0 = workstation slot number = shared resource processor CPU
1	sbDevice	13	device name string, preceded by 1 byte length
14	sbVolume	13	volume name string, preceded by 1 byte length. Length = 0 if volume is not mounted or device is not disk.

QueryDeviceName differs from QueryDeviceNames in that the former allows the caller to specify only a subset of the entries in the table.

Procedural Interface

QueryDeviceName (pbRouteSpec, cbRouteSpec, iDevice, pbNamesRet, cbNamesMax, pcbNamesRet): ercType

where

pbRouteSpec
cbRouteSpec

describe the node name of a workstation or shared resource processor CPU that contains the device name table to be queried.

iDevice

is a counting number starting at 0 used to index the first entry in the range of name table entries to be returned. The table entries are in no particular order.

pbNamesRet
cbNamesMax

describe a memory area where the device name entries will be returned.

pcbNameRet

is the memory address of the word where the count of bytes in the name entries returned will be placed.

Request Block

scbNameRet is always 2.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	1
3	nRespPbCb	1	2
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	346
12	iDevice	2	
14	reserved	4	
18	pbRouteSpec	4	
22	cbRouteSpec	2	
24	pbNamesRet	4	
28	cbNamesMax	2	
30	pcbNamesRet	4	
34	scbNamesRet	2	2

This page intentionally left blank

QueryDeviceNames (pbRouteSpec, cbRouteSpec, pbNamesRet, cbNamesMax, pcbNamesRet): ercType

Description

QueryDeviceNames returns the workstation or shared resource processor CPU device name table. The operation QueryDeviceNames is similar to QueryDeviceName except the latter allows the caller to specify that only a subset of the table entries be returned.

Procedural Interface

QueryDeviceNames (pbRouteSpec, cbRouteSpec, pbNamesRet, cbNamesMax, pcbNamesRet): ercType

where

pbRouteSpec
cbRouteSpec

describe the node name of a workstation or shared resource processor CPU that contains the device name table to be queried.

(iDevice)

is 0 by default if the procedural interface is used. *iDevice* is a counting number, which indexes the first name table entry to be returned. If the caller builds the request block and issues the request, the caller can specify a different index value for this parameter in the request block.

pbNamesRet
cbNamesMax

describe a memory area where the device name table is returned. (See QueryDeviceName for the format of each name table entry.)

pcbNamesRet

is the memory address of the word where the count of bytes in the name table will be placed.

Request Block

scbNamesRet is always 2.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	1
3	nRespPbCb	1	2
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	340
12	iDevice	2	
14	reserved	4	
18	pbRouteSpec	4	
22	cbRouteSpec	2	
24	pbNamesRet	4	
28	cbNamesMax	2	
30	pcbNamesRet	4	
34	scbNamesRet	2	2

QueryDiskGeometry (pbDevName, cbDevName, pbPassword, cbPassword, pDiskGeometryRet, sDiskGeometryRet): ercType

Description

QueryDiskGeometry returns disk geometry information in the memory area provided. The disk geometry information was previously set by a call to SetDiskGeometry.

Procedural Interface

QueryDiskGeometry (pbDevName, cbDevName, pbPassword, cbPassword, pDiskGeometryRet, sDiskGeometryRet): ercType

where

pbDevName
cbDevName

describe a character string containing the device or volume name. Square brackets surrounding the name are optional. The distinction between uppercase and lowercase is not significant in matching device names.

pbDevicePassword
cbDevicePassword

describe the device or volume password.

pDiskGeometryRet
sDiskGeometryRet

describe the memory area where the disk geometry information will be returned. The format of the returned information is shown below. If *sDiskGeometryRet* is not large enough, the information is truncated.

NOTE: On a physically-oriented disk (e.g., ST-506, SMD, and floppy), cylindersPerDisk, tracksPerCylinder, and sectors/Track all return nonzero values and reflect actual geometry. On a logically-oriented disk (such as a SCSI), cylindersPerDisk returns 0, and tracksPerCylinder and sectors/Track return nonzero, but they reflect a fictitious geometry that is useful to some internal procedures.

Offset	Field	Size (bytes)	Description
0	cylindersPerDisk	2	are the number of cylinders per disk.
2	tracksPerCylinder	1	are the number of tracks per cylinder
3	sectors/Track	1	are the number of sectors per track
4	bytesPerSector	2	are the number of bytes per disk sector
6	fRemovable	1	TRUE means the medium may be removed
7	fMountable	1	TRUE means the device is capable of having a valid CTOS file system
8	fUtilizeECC	1	TRUE means the device is configured to use error correction code to rectify read errors

Offset	Field	Size (bytes)	Description
9	fECCCapable	1	TRUE means the device is capable of using error correction code
10	fWritePrecompensation	1	TRUE means write precompensation is utilized by the disk
11	reserved	1	
12	sbResourceMgr	13	is an sb string containing the name of the service managing the disk
25	rgResourceId	5	is a array of five one-byte elements that contain physical address information for the disk
30	writePreCompCyl	2	is the starting cylinder for write precompensation current
32	gapLength	1	is the length of the unused area between sectors
33	stepRate	1	is an encoded number that represents the time interval between seek step pulses for disks that are controlled by a WD-1010 or WD-2010
34	spiralFactor	1	is the sector number offset between adjacent tracks for disks (typically SMD) that use a spiral layout of data sectors to improve performance
35	interleave	1	
36	blocksPerDisk	4	are the number of user addressable data blocks (usually 512 bytes) per disk

Offset	Field	Size (bytes)	Description
40	deviceClass	1	is the type of hardware controller. Each value indicates the following:

Value	Device Class
0	ST-506 disk devices controlled by a WD-1010/2010
1	floppy diskettes controlled by a WD-2797
2	Storage Module Disk (SMD) controller on a shared resource processor
3	reserved
4	SCSI devices controlled by an NCR 53C90
5	floppy diskettes controlled by a NEC 765
6	SCSI devices controlled by a Unisys ISK Gate Array
8	Memory Disk
10	Disk cache
12	SCSI devices controlled by an NCR 53C94

Offset	Field	Size (bytes)	Description
41	floppyDiskType	1	is the type of floppy disk. Each value indicates the following:
42	floppyDriveType	1	is the type of floppy drive. Each value indicates the following:

floppyDiskType:

Value	Device Class
0	Disk type unknown or no disk inserted.
1	Low capacity disk inserted.
2	High capacity disk inserted.

floppyDriveType:

Value	Device Class
0	5¼-inch drive controlled by a WD–2797.
1	5¼-inch drive controlled by a NEC 765.
2	3½-inch drive controlled by a NEC 765.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	2
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	366
12	reserved	6	
18	pbDevName	4	
22	sbDevName	2	
18	pbDevPassword	4	
22	sbDevPassword	2	
18	pDiskGeometryRet	4	
22	sDiskGeometryRet	2	

QueryExitRunFile (psbExitRunFileRet, ssbExitRunFileRetMax, psbPasswordRet, ssbPasswordRetMax, pPriorityRet): ercType

Description

QueryExitRunFile returns the name, password, and priority of the exit run file of the application partition. If executed in a system partition, QueryExitRunFile returns the name, password, and the global exit run file.

Procedural Interface

QueryExitRunFile (psbExitRunFileRet, sExitRunFileRetMax, psbPasswordRet, sPasswordRetMax, pPriorityRet): ercType

psbExitRunFileRet
sExitRunFileRetMax

define the memory area where the exit run file specification is returned. The first byte of the returned information is the size of the exit run file specification.

psbPasswordRet
sPasswordRetMax

define the memory area where the password for the exit run file is returned. The first byte of the returned information is the size of the password.

pPriorityRet

is the memory address of the word where the priority of the exit run file is returned.

Request Block

sPriorityRet is always 2.

Offset	Field	Size (Bytes)	Contents
	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	3
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	187
12	reserved	6	
18	psbExitRunFileRet	4	
22	sExitRunFileRetMax	2	
24	psbPasswordRet	4	
28	sPasswordRetMax	2	
30	pPriorityRet	4	
34	sPriorityRet	2	2

QueryFrameAttrs (iFrame, iCol, iLine, sBuff, pRetBuf, pcbRet): ercType

NOTE: This operation is supported on protected mode operating systems only.

Description

QueryFrameAttrs returns an attribute string in the virtual character map beginning at the specified coordinates of the specified frame. The operation provides line wraparound to return attribute strings existing in multiple lines.

Procedural Interface

QueryFrameAttrs (iFrame, iCol, iLine, sBuff, pRetBuf, pcbRet): ercType

where

iFrame

specifies the frame (byte). A value of 255 is treated as a special case and is reserved for internal use only.

iCol

specifies the horizontal coordinate (byte) within *iFrame* of the attribute string to be returned.

iLine

specifies the vertical coordinate (byte) within *iFrame* of the attribute string to be returned.

sBuff

is a word indicating the size in bytes of the string.

pRetBuf

is the memory address of the buffer to which the attribute string will be returned.

pcbRet

is the memory address of a word that contains the actual size of the string returned.

Request Block

scbRet is always 2.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	4
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	2
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	113
12	iFrame	1	
13	iLine	1	
14	iCol	1	
15	reserved	1	
16	pRetBuf	4	
20	sBuff	2	
22	pcbRet	4	
26	scbRet	2	

This page intentionally left blank.

QueryFrameBounds (iFrame, pcCols, pcLines): ercType

NOTE: This operation is supported on protected mode operating systems only.

Caution: *This operation works on workstation hardware only. Do not use it on shared resource processor hardware.*

Description

QueryFrameBounds returns the size in number of columns and lines for the specified frame.

Procedural Interface

QueryFrameBounds (iFrame, pcCols, pcLines): ercType

where

iFrame

specifies the frame (word) for which the size is to be returned. A value of 255 is treated as a special case and is reserved for internal use only.

pcCols

is the memory address of the word where the number of columns is to be returned.

pcLines

is the memory address of the word where the number of lines is to be returned.

Request Block

QueryFrameBounds is a system-common procedure.

QueryFrameChar (iFrame, iCol, iLine, pbRet): ercType

Description

QueryFrameChar returns a single character located in the character map at the specified coordinates of the specified frame.

Procedural Interface

QueryFrameChar (iFrame, iCol, iLine, pbRet): ercType

where

iFrame

specifies the frame (word). If *iFrame* = 0FFh, the coordinates *iCol* and *iLine* refer to virtual screen coordinates. This works exactly the same as if the caller has a frame that overlays the entire screen.

iCol
iLine

specify the horizontal and vertical coordinates (words) within *iFrame* of the character to be returned.

pbRet

is the memory address of the byte where the character is to be returned.

Request Block

QueryFrameChar is a system–common procedure.

QueryFrameCharsAndAttrs (iFrame, iCol, iLine, pbText, cbText, pbAttrs, cbAttrs): ercType

NOTE: This operation is supported on protected mode operating systems only. Applications that need to execute on real mode operating systems should use QueryCharsAndAttrs.

Description

QueryFrameCharsAndAttrs returns a character string and its associated attributes from the character map at the specified coordinates.

Procedural Interface

QueryFrameCharsAndAttrs (iFrame, iCol, iLine, pbText, cbText, pbAttrs, cbAttrs): ercType

where

iFrame

specifies either a frame or the virtual screen (word). If *iFrame* = 0FFh, then the coordinates *iCol* and *iLine* refer to virtual screen coordinates. This works exactly the same as if the caller has a frame that overlays the entire screen.

iCol
iLine

specify the horizontal and vertical coordinates (words) at which the first character is to be returned.

pbText
cbText

describe the character buffer where the text string is to be returned.

pbAttrs
cbAttrs

describe the attribute data buffer where the attributes associated with the text string will be returned.

NOTE: If cbText <> 0 and cbAttrs <> 0, then cbAttrs must equal cbText. Setting cbText = 0 allows the query of attributes only. Likewise if cbText <> 0 and cbAttrs = 0, then this is a query of characters only.

Request Block

QueryFrameCharsAndAttrs is a system-common procedure.

QueryFrameCursor (iFrame, piCol, piLine): ercType

NOTE: This operation is supported on protected mode operating systems only.

Description

QueryFrameCursor returns the cursor position for the specified frame.

Procedural Interface

QueryFrameCursor (iFrame, piCol, piLine): ercType

where

iFrame

> specifies the frame (word) where the cursor is to be returned. If *iFrame* = 0FFh, the coordinates returned at *piCol* and *piLine* refer to virtual screen coordinates. This works exactly the same as if the caller has a frame that overlays the entire screen.

piCol
piLine

> specify the memory addresses where the horizontal and vertical coordinates of the cursor within the frame will be returned.

Request Block

QueryFrameCursor is a system-common procedure.

This page intentionally left blank.

QueryFrameString (iFrame, iCol, iLine, sBuff, pRetBuf, pcbRet): ercType

NOTE: This operation is supported on protected mode operating systems only.

Description

QueryFrameString returns a character string in the virtual character map beginning at the specified coordinates of the specified frame. The operation provides line wraparound to return character strings existing in multiple lines.

Procedural Interface

QueryFrameString (iFrame, iCol, iLine, sBuff, pRetBuf, pcbRet): ercType

where

iFrame

specifies the frame (byte). A value of 255 is treated as a special case and is reserved for internal use only.

iCol

specifies the horizontal coordinate (byte) within *iFrame* of the character string to be returned.

iLine

specifies the vertical coordinate (byte) within *iFrame* of the character string to be returned.

sBuff

is a word indicating the size in bytes of the string.

pRetBuf

is the memory address of the buffer to which the character string will be returned.

pcbRet

is the memory address of a word that contains the actual size of the string returned.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	4
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	2
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	114
12	iFrame	1	
13	iLine	1	
14	iCol	1	
15	reserved	1	
16	pRetBuf	4	
20	sBuff	2	
22	pcbRet	4	
26	scbRet	2	

This page intentionally left blank.

QueryIOOwner (userNum, pBufferRet, sBufferMax, psDataRet): ercType

NOTE: This operation only works on virtual memory operating systems and is for use by programs executing in protected mode.

Description

QueryIOOwner identifies the processes that have exclusive access rights to specific ranges of I/O addresses. This operation returns a series of records, each of which contains a user number and its associated range of addresses. If the buffer at *pBufferRet* is too small, QueryIOOwner returns as many records as possible.

An application can use QueryIOOwner to identify unacquired I/O addresses. By calling SetIOOwner, the application can then obtain access rights to these addresses.

QueryIOOwner, in combination with SetIOOwner, is primarily used by the PC Emulator to prevent multiple processes from simultaneously using the same I/O addresses.

Procedural Interface

QueryIOOwner (userNum, pBufferRet, sBufferMax, psDataRet): ercType

where

userNum

is the user number whose I/O ownership is to be queried. If *userNum* is FFFFh, the I/O ownership for all user numbers is queried.

pBufferRet

is the memory address of the buffer where the records are returned. Each record has the following format:

Offset	**Field**	**Size (Bytes)**	**Description**
0	qFirstIoAddr	4	The first address in the range of I/O addresses.
4	cIoAddr	4	The count of addresses in the range of I/O addresses. Each address corresponds to one byte in memory.
8	userNum	2	The user number of the process that owns the range of I/O addresses.

sBufferMax

is the maximum size, in bytes, of the buffer at *pBufferRet*.

psDataRet

is the memory address of the word where the number of bytes returned is written.

Request Block

ssDataRet is always 2.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	2
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	466
12	userNum	2	
14	reserved	4	
18	pBufferRet	4	
22	sBufferMax	2	
24	psDataRet	4	
28	ssDataRet	2	2

This page intentionally left blank.

QueryKbdLeds: (pLedsRet): ercType

Caution: *This operation works on workstation hardware only. Do not use it on shared resource processor hardware.*

Description

QueryKbdLeds returns the state (on/off) of the keyboard LEDs.

Procedural Interface

QueryKbdLeds: (pLedsRet): ercType

where

pLedsRet

is the memory address of a byte (on CTOS operating systems) and a word (on BTOS systems) where state of the LEDs is returned. If a bit is set, the corresponding LED is on. The bits correspond to the LED keys as shown below:

Bit	**Key**
0 (low)	SCROLL LOCK
1	NUM LOCK
2	CAPS LOCK
3	f3
4	f2
5	f1
6	LOCK
7	OVERTYPE

Bit	Key
8	LTAI
9	ENQ
10	LOCAL
11	RCB
12	XMT

NOTE: Bits 8 through 12 are valid only on K3, K5, and SG101-K keyboards.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	55
12	reserved	6	
18	pLedsRet	4	
22	sLedsRet	2	*

*Is 1 on CTOS operating systems and 2 on BTOS systems

QueryKbdState (pKbdDescRet): ercType

Caution: *This operation works on workstation hardware only. Do not use it on shared resource processor hardware.*

Description

QueryKbdState returns the status of the Keyboard Process and the System Input Process to the structure provided by the caller.

Procedural Interface

QueryKbdState (pKbdDescRet): ercType

where

pKbdDescRet

is the memory address of a 16-byte keyboard descriptor area to which the status of the Keyboard Process and the System Input Process are returned. This structure is described below:

Offset	Field	Size (bytes)	Description
0	fUnencodedMode	1	Is character mode if FALSE or unencoded mode if TRUE.
1	sysInMode	1	A value of 0 is normal mode (neither playback nor recording mode is active).
			A value of 1 is recording mode (a copy of keyboard input is being written to the recording file specified by *fhSysIn*).

Offset	Field	Size (bytes)	Description
			A value of 2 is playback mode (input is being read from the submit file specified by *fhSysIn*). A value of 3 is escaped playback mode (the submit file specified by *fhSysIn* is in use, but a read-direct escape sequence was read from that file causing input to be read directly from the keyboard until a special key is pressed).
2	fhSysIn	2	Is the file handle of the currently open submit or recording file. If *sysInMode* is 0 (normal mode), *fhSysIn* is not meaningful.
4	reserved	12	

Request Block

sKbdDescRet is always 16.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	58
12	reserved	6	
18	pKbdDescRet	4	
22	sKbdDescRet	2	16

QueryLdtR (userNum, pSgLdtrRet): ercType

Description

QueryLdtR returns the selector that identifies the user number's Local Descriptor Table (LDT), provided the user number has an associated LDT. Otherwise, QueryLdtR returns the null selector.

Procedural Interface

QueryLdtR (userNum, pSgLdtrRet): ercType

where

userNum

is the user number returned from a CreatePartition or a GetPartitionHandle operation. If *userNum* is 0, the *userNum* of the calling program is used.

pSgLdtrRet

is the memory address of a 2–byte memory area where the LDT or null selector is returned.

Request Block

QueryLdtR is a system–common procedure.

This page intentionally left blank

QueryMail (pNMessagesRet, pIPriorityRet): ErcType

Description

QueryMail can be called by any program to display the fact that new mail is waiting for the user.

Prior to calling QueryMail, the mail user for which the new mail status is desired must be registered with the Mail Service. This is usually accomplished automatically during Executive signon, provided that the following entries have been included in the .user file:

:MailUserName:

:MailCenterName:

:MailPassword:

Registration with the Mail Service also happens when the user signs on to CT-MAIL or OFIS Mail.

Registration remains in effect until the user invokes the Signoff command in CT-MAIL or OFIS Mail, signs off the Executive, or reboots the workstation.

Procedural Interface

QueryMail (pNMessagesRet, pIPriorityRet): ErcType

where

pNMessagesRet

is the memory address of the word to which the number of new messages waiting for the user is returned.

pIPrecedenceRet

is the memory address of the word to which the precedence number of the highest priority message waiting for the user is returned. Precedence numbers are as follows:

Number	Meaning
0	Low
1	Normal
2	Urgent

Request Block

QueryMail is an object module procedure.

QueryMemAvail (pcParagraphRet): ercType

Description

QueryMemAvail returns the size (in 16-byte paragraphs) of all available free memory in an application partition.

The following operations allocate free memory:

AllocAllMemorySL
AllocAreaSL
AllocMemoryLL
AllocMemorySL
ExpandMemoryLL
ExpandAreaSL

Programs that operate in both real and protected modes should use QueryBigMemAvail rather than this operation because QueryMemAvail can report a maximum size of only 1M byte.

Procedural Interface

QueryMemAvail (pcParagraphRet): ercType

where

pcParagraphRet

is the memory address of a word to which the count of bytes available (divided by 16) is returned.

Request Block

scParagraphMax is always 2.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	48
12	reserved	6	
18	pcParagraphRet	4	
22	scParagraphMax	4	2

QueryModulePosition (pPosRet, bus, typeCode, iModule): ercType

Caution: *QueryModulePosition does not report the position of X-Bus+ cartridges.*

Description

QueryModulePosition determines the bus position of a workstation module. The bus (X-Bus or I-Bus), type code, and module number (if there is more than one module of the same type) are specified and the position is returned. (For details, see "X-Bus Management" in the *CTOS Operating System Concepts Manual.*)

Status code 35 ("No such module") is returned if the specified module does not exist.

Use GetModuleId for a list of all modules on an X-Bus.

Procedural Interface

QueryModulePosition (pPosRet, bus, typeCode, iModule) : ercType

where

pPosRet

points to a word where the position on the X-Bus is returned.

bus

is either 1 indicating I-Bus or 2 indicating X-Bus.

typeCode

is the module type code, which resides in the high byte of the module ID. For example, 32h is the module type code for the Voice Processor module. (For details on module type codes, see Appendix G.)

iModule

is the module number, where 0 is the first module of the specified type code to the right of the CPU.

Request Block

QueryModulePosition is an object module procedure.

QueryNodeName (pbSbName, cbSbName): ercType

NOTE: It recommended that you use this object module procedure over the QueryNodeName request documented in earlier operating system manual editions. The request no longer has a direct procedural interface.

Description

QueryNodeName returns the node name of the local node where this procedure is called.

Procedural Interface

QueryNodeName (pbSbName, cbSbName): ercType

where

pbSbName
cbSbName

describe the 13–byte buffer into which the node name is returned. The first byte is the count of bytes in the name; the remaining bytes contain the name string.

Request Block

QueryNodeName is an object module procedure.

This page intentionally left blank

QueryPagingStatistics (userNum, pStats, sStats, pCbStatsRet): ercType

NOTE: This operation only works on virtual memory operating systems and is for use by programs executing in protected mode.

Description

QueryPagingStatistics returns paging statistics for a specified user number or for the paging service. (For details on paging, see the section entitled "Demand Paging," in the *CTOS Operating System Concepts Manual*.)

Procedural Interface

QueryPagingStatistics (userNum, pStats, sStats, pCbStatsRet): ercType

where

userNum

is a user number or is 0FFFFh (word). If a user number is specified, paging statistics for that user number are returned. If 0FFFFh is specified, system paging statistics are returned.

pStats

is the memory address of a buffer in the data space of the caller to which the paging statistics are returned. If a user number is specified, the information is returned in a User Paging Statistics Record. If 0FFFFh is specified, the information is returned in a System Paging Statistics Record. Each of these records is described below.

sStats

is the size (word) of the buffer for the paging statistics.

pCbStatsRet

is the memory address to which the count of bytes written is returned.

User Paging Statistics Record

Offset	Field	Size (Bytes)	Description
0	size	2	current size in bytes of the User Paging Statistics Record. The number changes each time the size of the statistics record changes.
2	version	2	version number of the statistics record. The number is incremented each time the format of the statistics record changes.
4	cFramesRMin	4	minimum number of frames required by this user. This number is specified by the operating system. The paging service will not permit the number of resident frames to fall below this minimum.
8	cFramesRMax	4	maximum number of frames that should be allocated to this user (specified by the operating system). In general, *cFramesRMax* is the same as *cFramesV* (below). The global paging policy permits an application to compete with all other applications and use as many real frames as there are available in the system. It is possible to override the global paging policy by specifying :fSuppressGlobalPolicy: in the configuration file. In this case, *cFramesRMax* comes from the partition size specified in CmConfig.sys. The paging service does not assign more than *cFramesRMax* frames to this user. There are times when the paging service will increase *cFramesRMax*.

User Paging Statistics Record *(continued)*

Offset	Field	Size (Bytes)	Description
			To prevent thrashing, the paging service assures that a user will always have at least three unlocked frames. If the number of unlocked frames ever falls below this minimum, the paging service increments *cFramesRMax* in order to allocate another frame to this user. When a frame is unlocked, the paging service returns the borrowed frame and restores *cFramesRMax.*
12	cFramesV	4	number of virtual pages assigned to this user.
16	cFramesVSwapFile	4	maximum number of swap file frames for this user. This number is smaller than *cFramesV* because it does not include frames that come from code files or frames in locked hypersegments.
20	cFramesR	4	number of frames currently allocated to the user. This value changes according to the application demands, increasing to *cFramesRMax* for active foreground applications and shrinking to *cFramesRMin* for idle background applications.
24	cFramesWS	4	number of frames accessed or locked with the last revolution of the clock hand. For details on the clock (page replacement) algorithm, see "Replacing Pages" in the section entitled "Demand Paging" in the *"CTOS Operating System Concepts Manual."*
28	cFramesWSMin	4	minimum *cFramesWS* ever encountered for this user.
32	cFramesWSMax	4	maximum *cFramesWS* ever encountered for this user.

User Paging Statistics Record (*continued*)

Offset	Field	Size (Bytes)	Description
36	cFramesLocked	4	number of frames currently locked for this user. The application has no direct way of locking pages in memory. However, the operating system can make calls to the paging service to cause pages to be locked in memory. There are three types of frame locking: request data, DMA data, and swap disable data. While a request is outstanding, the request block and memory addressed by request/respond pointers is prefaulted and locked in. This prevents system services from blocking on page faults on behalf of their clients. While certain of the thread status "don't swap" bits are set, all pages of the user are swapped in and locked. See the section on Locking Swap Disable Data.
40	cPageFaultsTotal	4	total number of page faults handled for this user since the application began.
44	cPageFaults-SharedPages	4	total number of page faults for shared pages. A *shared page* is a page that is shared by more than one user. If two instances of the Executive are running, for example, they both share the same code pages. DLLs are always shared pages.
48	cWaitTicksTotal	4	total number of realtime clock ticks spent waiting for page faults. Each tick is worth 100 milliseconds, or 1 tenth of a second. When the paging service begins to service a page fault, it notes the value of the realtime clock.

User Paging Statistics Record (*continued*)

Offset	Field	Size (Bytes)	Description
			When the fault has been handled and the application is ready to restart, the paging service notes the ending value of the realtime clock, calculates the elapsed time, and adds this increment to *cWaitTicksTotal* for the user.
52	cWaitTicks-SharedPages	4	total number of clock ticks spent waiting for shared page faults.
56	cPagesCleaned	4	total number of pages cleaned for this user. When a page selected for replacement is dirty, it must be "cleaned" or written to the backing store before it can be used. When the clock algorithm encounters a page to be cleaned, it is placed on the cleaning queue and the pointer is stepped to the next page.
60	cWaitsFor-CleanPage	4	number of times this user had to wait for a page to be cleaned before a page fault could be serviced.
64	cPagesPrefaulted	4	number of pages prefaulted. The prefault algorithm tries to read in at least *cPagesToPrefault*, up to a maximum of *cPagesToPrefaultMax*. The number of prefaulted pages can be specified as a system configuration file option. In general, *cPagesPrefaulted* is greater than the sum of *cPagesPrefaultedUsed* and *cPagesPrefaultedUnused* because some pages can still be marked not-present and prefaulted in the page table. This means that the pages have not been used and have not been taken for replacement pages by the clock algorithm.

User Paging Statistics Record (*continued*)

Offset	Field	Size (Bytes)	Description
68	cPagesPrefaulted-Used	4	number of *cPagesPrefaulted* that were used. When a page fault occurs for a prefaulted page, the page table entry is changed to present, and *cPagesPrefaultedUsed* is incremented.
72	cPagesPrefaulted-Unused	4	number of *cPagesPrefaulted* that were not used. Whenever the clock algorithm must reclaim a prefaulted page, *cPagesPrefaultedUnused* is incremented.
76	cPagesQueued-ForCleaning	4	number of pages currently in the cleaning queue.
80	cPagesBeing-Cleaned	4	number of pages currently being written to backing store.
84	cSwapFilePages	4	number of swap file pages currently allocated to this user.

System Paging Statistics Record

Offset	Field	Size (Bytes)	Description
0	size	2	total size in bytes of the System Paging Statistics Record.
2	version	2	version number of the statistics record.
4	cFramesFree	4	total number of free page frames in the system.
8	cSwapPagesFree	4	total number of free swap pages in the swap file.

System Paging Statistics Record *(continued)*

Offset	Field	Size (Bytes)	Description
12	cEscapeNodesFree	4	total number of free *escape nodes* in the system. Escape nodes are small memory blocks maintained on a linked list and allocated when information about a not-present page does not fit in a page table entry (for example information about fixup, shared, and alias pages). Initially, all leftover memory that cannot be used for page frames is used for escape nodes. If the paging service needs additional escape nodes, it steals a page frame and turns it into escape nodes.
16	cEscapeNodesUsed	4	number of escape nodes currently being used in the system. The total number allocated is the sum of *cEscapeNodesUsed* and *cEscapeNodesFree*.
20	escapeNodeSize	4	size in bytes of an escape node.
24	cFramesAllocated	4	total number of page frames in the system.
28	cPagesToPrefault	4	number of pages the prefault algorithm attempts to prefault. It tries to read at least *cPagesToPrefault*, up to a maximum of *cPagesToPrefaultMax*. This value can be configured in the system configuration file.
32	cPagesToPrefaultMax	4	maximum number of pages the prefault algorithm attempts to prefault. This value can be configured in the system configuration file.
36	cPageFaultsIntsDisabled	4	number of times the paging service encountered a page fault while interrupts are disabled.
40	cSwapFilePagesInitial	4	number of pages in the swap file when it was first opened.

System Paging Statistics Record *(continued)*

Offset	Field	Size (Bytes)	Description
44	cSwapFilePagesCurrent	4	current number of pages in the swap file. If the swap file was extended, *cSwapFilePagesCurrent* will be larger than *cSwapFilePagesInitial.*
48	cSwapFilePagesMax	4	the maximum size that the swap file can grow to. This number can be specified in the configuration file.
52	cSwapFileExtensions	4	number of times the swap file was extended.
56	cSwapFileExtension-Attempts	4	number of times an attempt was made to extend the swap file. If the swap file runs out of space and the disk is full, the paging service will periodically attempt to extend the swap file. If an application is killed because the swap file is full, this gives the user a chance to delete some files in order to allow the swap file to grow.
60	sbSwapFileName	84	a string that specifies the name of the swap file. The first byte contains the length of the string.
144	cFramesLockedTotal	4	number of total number of locked frames in the system.

Request Block

QueryPagingStatistics is a system-common procedure.

QueryProcessInfo (wProcessId, pInfoRet, sInfoRet): ercType

Caution: *This operation is not supported on shared resource processors.*

Description

QueryProcessInfo returns the priority, delta priority, and user number of either the caller or another process. In addition, it identifies the Task State Segment selector for the process, and returns the number of times the process has been suspended.

Status code 39 "(No such user number") is returned if *wUserNumber* is an invalid user number.

Procedural Interface

QueryProcessInfo (wProcessId, pInfoRet, sInfoRet): ercType

where

wProcessId

is the process ID of the process for which information is to be returned. QueryProcessInfo obtains the user number by calling QueryProcessNumber. If wProcessId is 0, QueryProcessInfo returns the process ID of the caller in the *processUserNumb* field at *pInfoRet*.

pInfoRet

is the memory address of a location to which the following process information is returned:

Offset	Field	Size (Bytes)	Description
0	priority	1	The priority of the process.
1	deltaPriority	1	The delta priority of the process.
2	reserved	2	
4	processUserNumb	2	The user number of the specified process. When the *wUserNumber* parameter is 0, this value is the user number of the caller.
6	sgTss	2	The selector of the Task State Segment for the process.
8	reserved	2	
10	cSuspends	2	The number of times the process has been suspended.

sInfoRet

is the size of the information returned.

Request Block

QueryProcessInfo is a Kernel primitive.

This page intentionally left blank.

QueryProcessNumber (pPidRet): ercType

Description

QueryProcessNumber returns a word value that uniquely identifies the calling process.

Procedural Interface

QueryProcessNumber (pPidRet): ercType

where

pPidRet

is the memory address of a word where the process ID of the inquiring process is returned.

Request Block

QueryProcessNumber is a system-common procedure.

This page intentionally left blank

QueryRequestInfo (rq, pStatusRet, sStatusRetMax): ercType

Description

QueryRequestInfo determines the exchange to which a request and its local service code are routed. On a shared resource processor, QueryRequestInfo reflects the status of requests that are being served on the local processor board only. Status code 31 ("No such request code") is returned if the request is not defined. (For details on defining requests, see the *CTOS Operating System Concepts Manual.*)

System services that filter or serve requests use this operation on every request they wish to serve. The caller should save away any existing information, that is the service exchange and local service code, so that this information can be restored before the caller deinstalls.

Procedural Interface

QueryRequestInfo (rq, pStatusRet, sStatusRetMax): ercType

where

rq

is the request code.

pStatusRet
sStatusRetMax

describe the memory area to which the status information is returned. The status information is a 4-byte structure defined as follows:

Offset	**Field**	**Size (Bytes)**
0	Service Exchange	2
2	Local Service Code	2
4	Shared resource processor Routing Type	1
5	Net Routing	1

Request Block

Offset	**Field**	**Size (Bytes)**	**Contents**
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	qCode	2	271
12	rq	2	
14	reserved	4	
18	pStatusRet	4	
22	sStatusRetMax	2	

QueryTrapHandler (iInt, pTrapInfoRet, sTrapInfoRet): ercType

Description

QueryTrapHandler returns the starting address of the current interrupt handler for the specified interrupt level. In addition, it returns the type of gate that corresponds to the handler, as well as a flag that indicates whether the handler is mediated or raw.

To assign a new handler for the interrupt, the application can use either SetTrapHandler or Set386TrapHandler. To assign the default handler for the interrupt, the application can use ResetDeviceHandler. (See SetTrapHandler, Set386TrapHandler, and ResetDeviceHandler.)

For details on interrupt handlers, see "Interrupt Handlers" in the *CTOS Operating System Concepts Manual.*

Procedural Interface

QueryTrapHandler (iInt, pTrapInfoRet, sTrapInfoRet): ercType

where

iInt

is the interrupt level (0 to 255).

pTrapInfoRet

is the address of a structure to which information about the current trap handler is returned. The structure has the following format:

Offset	**Field**	**Size (Bytes)**	**Description**
0	pIntHandler	4	The starting address of the current interrupt handler.

Offset	Field	Size (Bytes)	Description
4	gateType	1	The type of gate that corresponds to the current interrupt handler. Each value indicates the following:
			Value — **Gate Type**
			E5h — Task gate
			E6h — 286 interrupt gate
			E7h — 286 trap gate
			EEh — 386 interrupt gate
			EFh — 386 trap gate
5	fMediated	1	A flag that is TRUE (255) if the current handler is mediated. This flag is FALSE if the current handler is raw.

sTrapInfoRet

is the size of the structure returned at *pTrapInfoRet*. If *sTrapInfoRet* is 6, the entire structure is returned. If *sTrapInfoRet* is 4, only the *pIntHandler* field is returned.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	470
12	iInt	2	
14	reserved	4	
18	pTrapInfoRet	4	
22	sTrapInfoRet	2	

This page intentionally left blank

QueryUserLocation (userNum, pbInfoRet, cbInfoMax): ercType

NOTE: Programs that may run on earlier versions of the operating system should use the object module procedure GetUserLocation rather than this request to obtain the same information. By doing so, you will avoid application termination with status code 31 ("No such request code").

Description

QueryUserLocation returns information about the specified user number in the memory area provided. If the memory area is too small to contain all the information returned by this operation, the information is truncated.

The information returned by this operation can be used, for example, to determine if the user number is executing on a specified SRP processor board. (See the description of the related operation, QueryBoardInfo.)

Procedural Interface

QueryUserLocation (userNum, pbInfoRet, cbInfoMax): ercType

where

userNum

is the user number (word) returned from a CreatePartition or a GetPartitionHandle operation. If userNum is 0, the user number is that of the calling program.

pbInfoRet
cbInfoMax

describe the memory area at which the information described below is returned.

Offset	Field	Size (bytes)	Description
0	bCode	1	(See below.)
1	localUserNum	2	Is the local user number associated with the *iCpu* (described below).
3	iCpu	2	Is the number of the CPU upon which the specified user number is executing. (The user number consists of a *localUserNum* and an *iCpu*. For workstations, the value of *iCpu* is 0.)

bCode is one of the following values:

Value	Meaning
0	The user number is on this CPU.
1	The user number is on this shared resource processor (shared resource processors only).
2	The user number is on a cluster workstation.
3	The user number is located over the network.
4	The user number is not valid.

Request Block

QueryUserLocation is a system-common procedure.

QueryVidBs (pBswa, pBsVidStateRet): ercType

Description

QueryVidBs returns information about video byte streams to the client structure.

Procedural Interface

QueryVidBs (pBswa, pBsVidStateRet): ercType

where

pBswa

is the memory address of the same Byte Stream Work Area that was supplied to OpenByteStream.

pBsVidStateRet

is the memory address of a 16-byte structure with the following format:

Offset	Field	Size (bytes)	Description
0	nFrames	1	frame number
1	nFrameLines	1	number of lines in frame
2	nCols	1	number of columns in frame
3	lineNum	1	current line number
4	colNum		current column number
5	fCursurVisible	1	TRUE if cursor visible
6	fPauseEnabled	1	TRUE if pausing between full frames of text is enabled
7	charAttrMode	1	current character attribute mode, as specified in controlling character attributes escape sequence
8	fLiteralMode		is TRUE if in literal mode
9	reserved	7	

Request Block

QueryVidBs is an object module procedure.

QueryVideo (pBuffer, sBuffer): ercType

Caution: *This operation works on workstation hardware only. Do not use it on shared resource processor hardware.*

Description

QueryVideo places information describing the level of video capability of the workstation in the specified memory area. QueryVideo is an object module procedure that is similar to the QueryVidHdw request. Both operations return video capability information in the same format. (See QueryVidHdw for the format of the video structure.)

QueryVideo, however, checks the operating system version and fills in all fields in the video structure appropriately. QueryVidHdw fills in only certain fields according to the operating system version.

For example, if QueryVideo is called in a program running on the CTOS II 1.0 operating system, QueryVideo fills in the *wxAspect* and *wyAspect* fields with appropriate values for particular hardware. QueryVidHdw does not support these fields for CTOS II 1.0 and, therefore, would not fill them in. (For details, see QueryVidHdw.)

Procedural Interface

QueryVideo (pBuffer, sBuffer): ercType

where

pBuffer
sBuffer

describe the buffer to which the video capability information is to be copied. If *sBuffer* is too small, the data is truncated.

Request Block

QueryVideo is an object module procedure.

QueryVidHdw (pBuffer, sBuffer): ercType

Description

QueryVidHdw places information describing the level of video capability of the workstation in the specified memory area. When writing software that must work on several workstation models, use QueryVidHdw to determine the level of video capability present before calling the ResetVideo operation.

QueryVidHdw is similar to QueryVideo. Both operations return video capability information in the same format described below. QueryVidHdw, however, fills in certain fields only for the specified workstation operating system version. QueryVideo fills in all fields. (See QueryVideo for more information.)

The fields that are filled in by QueryVidHdw are listed following the video format below.

The format of the video information returned is as follows:

Offset	Field	Size (Bytes)	Contents
0	level	1	Level of video capability: 0 = reserved 1 = reserved 2 = reserved 3 = reserved 4 = workstations with a hardware character map 6 = workstations with a software bit map only
1	nLinesMax	1	Maximum number of lines that fill the entire screen for the current resolution.
2	nColsNarrow	1	Number of columns in narrow mode.
3	nColsWide	1	Number of columns in wide mode (for example, 132 for work-stations with 132 column capability and 146 for workstations with 146 column capability).
4	graphicsVersion	1	Level of bit map capability: 0 = none 1 = reserved 2 = for monochrome graphics--GC-001(B25-GRA)

Offset	Field	Size (Bytes)	Contents
			3 = for color graphics workstations-- GC-001(B25-GRA) or 3 = for hardware with enhanced video capability (BTOS VAM 3.1.1 and earlier) 5 = 1024x768 resolution (GC-003 with VM-003) 6 = 720x348 resolution (GC-003 with VM-001, B25-D1 or VM-002, B25-D2) 8 = resolutions other than 1024x768 or 720x348
5	nPixelsHigh	2	Number of pixels high for the graphics version of the bit map.
7	nPixelsWide	2	Number of pixels wide for the graphics version of the bit map.
9	saGraphicsBoard	2	Segment address of the memory segment that is assigned to the bit map.
11	ioPort	2	I/O port of the Graphics Board memory.
13	wBytesPerLine	2	Number of bytes in one raster line for a bit map video display.
15	nCharHeight	1	Character cell height in pixels.
16	nCharWidthNarrow	1	Character cell width in pixels in narrow mode.
17	nCharWidthWide	1	Character cell width in pixels in wide mode.

Offset	Field	Size (Bytes)	Contents
18	pBitmap	4	Memory address of the bit map structure. (See below for the format of the bit map structure.)
22	pFont	4	Memory address of the font currently being used by the video display. The font may be a character font on a character map workstation without graphics or a raster font on a workstation with a bit map display. (See the *Graphics Programmer's Guide* for more information on raster fonts.)
26	bModuleType	1	Number that is assigned to the module controlling the video display (for the X-Bus). See Appendix G for a list of the X-Bus and I-Bus Module ID assignments.
27	bModulePos	1	Number that specifies the position of the video display controlling the X-Bus module (for example, 1 indicates the first module to the right of the workstation).
28	wModuleEar	2	I/O port address that provides access to the video display controlling module.
30	nYCenter	2	Offset of the first character cell in a video display from the top of the display.
32	nXCenterNarrow	2	Offset of the first character cell in a video display from the left side of the display in narrow mode.

Offset	Field	Size (Bytes)	Contents
34	nXCenterWide	2	Offset of the first character cell in a video display from the left side of the display in wide mode.
36	wxAspect	2	Ratio between x and y physical screen dimensions. One value is 32768. The other value is less than 32768.
	wyAspect	2	
40	nPlanes	1	Number of graphics planes.
41	fColorMonitor	1	Flag. If TRUE, the color monitor is supported.
42	nColors	2	Number of colors supported.
44	fBackgroundColor	1	Flag. TRUE means background color capability.
45	fHardwareCharMap	1	Flag. TRUE means a character-mapped workstation.
46	nAlternateLinesMax	2	Maximum number of lines that fill the entire screen for the current resolution. This value is provided for monitors and video controllers capable of changing the number of lines. (For example, workstations with Enhanced Video (EV) can change the number of lines from 29 to 34.)
48	nAlternateCharHeight	2	Character cell height in pixels for workstations with the ability to change character height. (For example, workstations with EV have this ability.)
50	wVidRelease	2	Video software release number.
52	wVidVersion	2	Video software internal version number.

Offset	Field	Size (Bytes)	Contents
54	fGenericColor	1	Flag. TRUE means the hardware supports the single-palette format returned by the ProgramColorMapper operation. (For details, see "Using Color" in the *CTOS Programming Guide*.
55	fVariableCharMap	1	Flag. TRUE means the resolution of the display will change when there is a change in the line mode and the hardware reformats (alters the size of) the character map.
56	fMultiSystemFonts	1	Flag. TRUE means more than one system font is supported.
57	fUseWsLine	1	Flag. TRUE means use wsLine in the Video Control Block (VCB). FALSE means use sLine instead. (See "Video Control Block" in the Chapter 4, "System Structures.")
58	wLevelsPerPrimary	2	Maximum number of color levels per primary color (red, green, or blue) offered by a color-mapped graphics controller in combination with the monitor to which it is connected.
60	fWhiteBackground	1	Flag. TRUE means the default background screen is white (systems with color mapper capability only).
61	nExtraRasterLines	2	Number of raster lines below the display bit map.
63	fStdVgaCharAttributes	1	Flag. TRUE means an EISA/ISA Bus system with no add-on CTOS VideoCard; 0 means all other systems.

The following fields (indicated by their offsets) are filled in by QueryVidHdw for each operating system version:

Offsets	CTOS Version	BTOS Version
0-11	CTOS 9.9 and earlier	BTOS 8.0 and earlier; real mode; BTOS II versions earlier than 3.2
0-38	CTOS II 1.0	
0-44	CTOS/VM 2.0	BTOS II 1.0
45-54	CTOS/VM 2.2	BTOS II 2.0
55-61	CTOS/XE 3.0	

(For details on release numbers, see CurrentOsVersion.)

The format of the Bitmap field in the video structure above is as follows:

Offset	Field	Size (Bytes)	Contents
0	cbLineBm	2	Same as sBytesPerLine.
2	nWidthBm	2	Same as nPixelsWide.
4	nHeightBm	2	Same as nPixelsHigh.
6	fScreenBm	1	Flag that is always TRUE.
7	bRegBase	1	Same as upper byte of ioPort.
8	bPlanes	1	Number of planes in bitmap.
9	bSegs	1	Always 1.
10	wSegHeight	2	Same as nHeightBm.
12	pBm	4	Physical bitmap address.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	21
12	reserved	6	
18	pBuffer	4	
22	sBuffer	2	

QueryWsNum (pWsNumRet): ercType

Caution: *This operation is for compatibility with prior operating system releases only and may not be available in future releases. It is not supported on shared resource processors or on workstations attached to shared resource processor servers.*

Description

QueryWsNum returns the workstation identification number for the specified cluster workstation. This number can be used as a parameter to the GetWsUserName operation.

Procedural Interface

QueryWsNum (pWsNumRet): ercType

where

pWsNumRet

is the memory address of a word where the cluster workstation identification number is returned.

On a shared resource processor server, bits 0–6 of *WsNumRet* contain the Cluster Processor (CP) board number. Bits 7–15 contain the workstation number on the CP board.

On all other servers, *WsNumRet* contains the workstation number.

Request Block

sWsNumRet is always 2.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	61
12	reserved	6	
18	pWsNumRet	4	
22	sWsNumRet	2	2

QueryZoomBoxPosition (piColRet, piLineRet): ercType

Description

QueryZoomBoxPosition is used by an application program to determine the location of the upper left corner of a box created by the ZoomBox operation. The information returned can then be used to plot text within the box. Note that the coordinates of the actual box edge are returned. To plot text inside the box, increase each of these values as necessary.

Procedural Interface

QueryZoomBoxPosition (piColRet, piLineRet): ercType

where

piColRet

> is the memory address of a word where the column number of the box is returned. If no box has been displayed, the value 0FFFFh will be returned.

piLineRet

> is the memory address of a word where the line number of the box is returned. If no box has been displayed, the value 0FFFFh will be returned.

Request Block

QueryZoomBoxPosition is an object module procedure.

This page intentionally left blank

QueryZoomBoxSize (nCols, nLines, psDataRet): ercType

Description

QueryZoomBoxSize determines the amount of memory required to save the contents of a zoom box.

Procedural Interface

QueryZoomBoxSize (nCols, nLines, psDataRet): ercType

where

nCols

is the number of columns inside the box.

nLines

is the number of lines (rows) inside the box.

psDataRet

is the memory address of a word to which the size (in bytes) of the box is returned.

Request Block

QueryZoomBoxSize is an object module procedure.

This page intentionally left blank

QueueMgrVersion (pVerStruct, sVerStruct, pcbRet): ercType

Description

QueueMgrVersion returns a structure that identifies the version of the Queue Manager, indicates if a cache is to be used, and specifies the maximum number of dynamic queues available.

Procedural Interface

QueueMgrVersion (pVerStruct, sVerStruct, pcbRet): ercType

where

pVerStruct
sVerStruct

describe the memory area to which the structure containing the version and installation parameters is written. The format of the structure is as follows:

Offset	**Field (Bytes)**	**Size**	**Description**
0	version	2	internal version identifier, which changes with each release of the Queue Manager.
2	fUseCache	1	corresponds to the user input field in the command form of the Install Queue Manager command.

Offset	Field (Bytes)	Size	Description
3	nCacheSlots	2	corresponds to the maximum number of dynamic queues in the Install Queue Manager command.

pcbRet

describes a word into which the length of the returned structure is placed.

Request Block

scbRet is always 2.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	1
3	nRespPbCb	1	2
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	8210
12	reserved	6	
18	pVerStruct	4	
22	sVerStruct	2	
24	pcbRet	4	
28	scbRet	2	2

Read (fh, pBufferRet, sBufferMax, lfa, psDataRet): ercType

Description

Read transfers an integral number of 128-, 256-, 512-, or 1024-byte sectors from disk to memory. Read returns only when the requested transfer is complete. The ReadAsync and CheckReadAsync operations are used to overlap computation and input/output transfer.

To accommodate programming languages in which Read is a reserved word, ReadFile is permitted as a synonym for Read.

Procedural Interface

Read (fh, pBufferRet, sBufferMax, lfa, psDataRet): ercType

where

fh

is a file handle returned from an OpenFile operation. The device can be open in either read or modify mode.

pBufferRet

is the memory address of the first byte of the buffer where the data is to be read. The buffer must be word aligned.

sBufferMax

is the count of bytes to be read to memory. It must be a multiple of the sector size.

lfa

is the byte offset, from the beginning of the file, of the first byte to be read. It must be a multiple of the sector size.

psDataRet

is the memory address of the word where the count of bytes successfully read is to be returned.

Request Block

ssDataRet is always 2.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	2
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	35
12	fh	2	
14	lfa	4	
18	pBufferRet	4	
22	sBufferMax	2	
24	psDataRet	4	
28	ssDataRet	2	2

ReadActionCode (pCodeRet): ercRet

Description

ReadActionCode returns the action code, if any, and resets a flag indicating that an action code is available. If no action code is available, ReadActionCode returns status code 609 ("No action code available"). All version 3.3 operating systems and later return the raw unencoded value generated by the attached hardware if the application is operating in raw unencoded mode. It returns the emulated value if the application is operating in character mode or unencoded mode. In BTOS, or if the application is operating in BTOS mode, the raw value is always returned. For the raw keyboard code values, see Table C-2, "Keyboard Codes: Key Differences for K2, K3, K5, and SG101-K Keyboards."

Under BTOS, the raw value was always returned. To run an application linked originally under BTOS that uses ReadActionCode without modifying the program source, use the **Set Keyboard Info** command to set the run file in BTOS mode. In BTOS mode, the raw value is always returned. For more information, see your Software Release Announcement.

Procedural Interface

ReadActionCode (pCodeRet): ercRet

where

pCodeRet

is the memory address of a byte where the keyboard code of the key that was depressed while the ACTION key was depressed is to be returned.

Request Block

sCodeRet is always 1.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	60
12	reserved	6	
18	pCodeRet	4	
22	sCodeRet	2	1

ReadActionKbd (fGlobal, pActionCharRet): ercType

Description

ReadActionKbd detects ACTION key sequences. When it detects the depression of the ACTION key, ReadActionKbd returns the keyboard code of the key pressed in conjunction with the ACTION key. All version 3.3 operating systems and later return the raw unencoded value generated by the attached hardware if the application is operating in raw unencoded mode. It returns the emulated value if the application is operating in character mode or unencoded mode. In BTOS, or if the application is operating in BTOS mode, the raw value is always returned. For the raw keyboard code values, see Table C-2, "Keyboard Codes: Key Differences for K2, K3, K5, and SG101-K Keyboards."

Under BTOS, the raw value was always returned. To run an application linked originally under BTOS that uses ReadActionCode without modifying the program source, use the **Set Keyboard Info** command to set the run file in BTOS mode. In BTOS mode, the raw value is always returned. For more information, see your Software Release Announcement.

Procedural Interface

ReadActionKbd (fGlobal, pActionCharRet): ercType

where

fGlobal

is a flag for reporting ACTION keys. If TRUE, then all ACTION key sequences at the workstation are reported. If FALSE, then ACTION keys are reported only when the keyboard has been assigned to the application.

pActionCharRet

is the address of a 1-byte memory location where the key code is copied.

Request Block

sActionCharRet is always 1.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	256
12	fGlobal	1	
13	reserved	5	
18	pActionCharRet	4	
22	sActionCharRet	2	1

ReadAsync (fh, pBufferRet, sBufferMax, lfa, pRq, exchangeReply): ercType

Description

ReadAsync initiates the transfer of an integral number of 512-byte sectors from disk to memory. CheckReadAsync must be called to check the completion status of the transfer.

The information returned by Read with its *psDataRet* argument and ercType status is obtained by CheckReadAsync.

Procedural Interface

ReadAsync (fh, pBufferRet, sBufferMax, lfa, pRq, exchangeReply): ercType

where

fh

is a file handle returned from an OpenFile operation. The file can be open in either read or modify mode.

pBufferRet

is the memory address of the first byte of the buffer where the data is to be read. The buffer must be word aligned.

sBufferMax

is the count of bytes to be read to memory. It must be a multiple of 512.

lfa

is the byte offset, from the beginning of the file, of the first byte to be read. It must be a multiple of 512.

pRq

is the memory address of a 64-byte area to be used as workspace by ReadAsync.

exchangeReply

is an exchange provided by the client for the exclusive use of ReadAsync and CheckReadAsync.

Request Block

ReadAsync and CheckReadAsync are procedural interfaces to read. (See Read.)

ReadBsRecord (pBswa, pBufferRet, sBufferMax, psDataRet): ercType

Description

ReadBsRecord reads the specified count of bytes from the open input byte stream identified by the memory address of the Byte Stream Work Area to the specified memory area. ReadBsRecord always reads the count of bytes specified except when fewer than that count remain in the file or when an input/output error occurs. If fewer than the specified count of bytes (or no bytes) remain in the file, status code 1 ("End of file") is returned in conjunction with the actual count of bytes read.

Procedural Interface

ReadBsRecord (pBswa, pBufferRet, sBufferMax, psDataRet): ercType

where

pBswa

is the memory address of the same Byte Stream Work Area that was supplied to OpenByteStream.

pBufferRet

is the memory address of the first byte of the buffer where the data is to be read.

sBufferMax

is the count of bytes to be read to memory.

psDataRet

is the memory address of the word where the count of bytes successfully read is returned.

Request Block

ReadBsRecord is an object module procedure.

ReadByte (pBswa, pbRet): ercType

Description

ReadByte reads one byte from the open input byte stream identified by the memory address of the Byte Stream Work Area. If no bytes remain in the file, status code 1 ("End of file") is returned.

Procedural Interface

ReadByte (pBswa, pbRet): ercType

where

pBswa

is the memory address of the same Byte Stream Work Area that was supplied to OpenByteStream.

pbRet

is the memory address of the byte where the data is returned.

Request Block

ReadByte is an object module procedure.

This page intentionally left blank

ReadBytes (pBswa, cbMax, pPbRet, pcbRet): ercType

Description

ReadBytes reads up to the specified count of bytes from the open input byte stream identified by the memory address of the Byte Stream Work Area. The count of bytes made available by this operation is chosen to optimize hardware performance and is not predictable. It can range from one to the specified maximum.

ReadBytes returns the memory address of the data bytes in its buffer rather than moving the data to a specified location. This optimizes performance, but imposes the restriction that the calling process must completely process the data before calling ReadBytes again. If this restriction is inconvenient, the ReadBsRecord operation should be used instead. If no bytes remain in the file, status code 1 ("End of file") is returned.

Procedural Interface

ReadBytes (pBswa, cbMax, pPbRet, pcbRet): ercType

where

pBswa

is the memory address of the same Byte Stream Work Area that was supplied to OpenByteStream.

cbMax

is the maximum count of bytes of data that the calling process will accept.

pPbRet

is the memory address to which the memory address of the data is to be returned.

pcbRet

is the memory address of a word to which the actual count of data bytes made available is to be returned.

Request Block

ReadBytes is an object module procedure.

ReadByteStreamParameterC (pBs, wParamNum, pwParamRet): ercType

Description

ReadByteStreamParameterC reports the current value of the specified communications line parameter.

(See "Communications Programming" in the *CTOS Operating System Concepts Manual.)*

The value of each ParamNum is as follows:

Code	Description
0	Number of data bits (5-8)
1	Parity (0=none, 1=even, 2=odd, 3=one, 4=zero)
2	Baud Rate (word) (truncated to nearest legal value)
3	Stop Bits (1=1, 2=2)
4	Transmit time out (in seconds) (0000h=immediate) (FFFFh=infinite)
5	Receive time out (in seconds) (0000h=immediate) (FFFFh=infinite)
6	Carriage return/line feed mapping mode (0=binary, 1=new line)
7	New Line mapping mode (0=binary, 1=CR, 2=CR/LF)
8	Line Control (0=none, 1=XON/XOFF, 2=CTS, 3=both)
9	EOF byte (low byte is EOF byte, high byte is TRUE when EOF checking is enabled, FALSE otherwise)
10	Receive baud rate (word)
11	Transmit baud rate (word)
12	Tab Expansion Size (byte; 0=treat tabs literally)
13	Maximum Line Length (byte; for auto-line-wrap; 0=disabled)

ParamNum 2 returns status code 7 ("Not implemented") when receive and transmit baud rates are not identical (and both transmit and receive clocks are actually in use).

ParamNum 10 or 11 can be used to query receive and transmit baud rates separately.

Procedural Interface

ReadByteStreamParameterC (pBs, wParamNum, pwParamRet): ercType

where

pBs

is the memory address of the Byte Stream Work Area (BSWA).

wParamNum

is the parameter number to read (word).

pwParamRet

is the memory address to which the word value of the specified parameter is returned.

Request Block

ReadByteStreamParameterC is an object module procedure.

ReadCommLineStatus (commLineHandle, wStatusMask, pwStatus): ercType

Description

ReadCommLineStatus allows certain RS-232 signals whose function is not defined by the the serial communications controller (such as the 8274 controller) to be queried by the application program in a machine-independent fashion.

ReadCommLineStatus reads the values of the specified status bits.

wStatusMask is a word with the following bit masks:

Mask	**Description**
0001h	SCD (Secondary Carrier Detect)
0002h	SCTS (Secondary Clear-to-Send)
0004h	SRX (Secondary Receive)
0008h	DSR (Data Set Ready)
0010h	RNG (Ring Indicator)
0020h	SQ (Signal Quality)

Masks may be ORed together to reference more than one status bit at a time. Only the status bits selected by *wStatusMask* are accessed. The others are not read.

Procedural Interface

ReadCommLineStatus (commLineHandle, wStatusMask, pwStatus): ercType

where

commLineHandle

is a handle returned by InitCommLine.

wStatusMask

is a bit mask that selects the status lines to read.

pwStatus

is the address of a location in memory where selected bits are returned.

Request Block

ReadCommLineStatus is a system–common procedure.

*ReadDaFragment (pDawa, qiRecord, pFragmentRet, rbFragment,
cbFragment): ercType*

Description

ReadDaFragment reads a record fragment from the open DAM file identified by the memory address of the Direct Access Work Area. The returned record fragment is specified by the record number, relative offset, and byte count.

Procedural Interface

*ReadDaFragment (pDawa, qiRecord, pFragmentRet, rbFragment,
cbFragment): ercType*

where

pDawa

is the memory address of the same Direct Access Work Area that was supplied to OpenDaFile.

qiRecord

is a 32-bit unsigned integer specifying the number of the record containing the record fragment to be read. *qiRecord* must correspond to an existing record.

pFragmentRet

is the memory address of the memory area where the record fragment is returned.

oFragment

is the offset from the beginning of the record to the first byte of the record fragment.

cbFragment

is the size of the record fragment.

Request Block

ReadDaFragment is an object module procedure.

ReadDaRecord (pDawa, qiRecord, pRecordRet): ercType

Description

ReadDaRecord reads a record from the open DAM file identified by the memory address of the Direct Access Work Area. The returned record is specified by the record number.

Procedural Interface

ReadDaRecord (pDawa, qiRecord, pRecordRet): ercType

where

pDawa

is the memory address of the same Direct Access Work Area that was supplied to OpenDaFile.

qiRecord

is a 32-bit unsigned integer specifying the number of the record to be read. *qiRecord* must correspond to an existing record.

pRecordRet

is the memory address of the memory area where the record is returned.

Request Block

ReadDaRecord is an object module procedure.

This page intentionally left blank

ReadDirSector (pbDirSpec, cbDirSpec, pbPassword, cbPassword, iSector, pBufferRet): ercType

Description

ReadDirSector reads a 512-byte sector of the specified directory. ReadDirSector is used primarily by the Executive.

Procedural Interface

ReadDirSector (pbDirSpec, cbDirSpec, pbPassword, cbPassword, iSector, pBufferRet): ercType

where

pbDirSpec
cbDirSpec

describe a character string of the form {Node}[VolName]DirName.

pbPassword
cbPassword

describe parameter values that are ignored. Use zeros for these parameters.

iSector

is the number of the sector to be read within the directory.

pBufferRet

is the memory address of the first byte of the buffer where the data is to be read. The buffer must be word aligned.

Request Block

sBufferMax is always 512.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	2
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	25
12	reserved	2	
14	iSector	2	
16	reserved	2	
18	pbDirSpec	4	
22	cbDirSpec	2	
24	pbPassword	4	
28	cbPassword	2	
30	pBufferRet	4	
34	sBufferMax	2	512

ReadHardId (pId): ercType

NOTE: Applications that need to run on older BTOS operating systems should use the OldReadHardId operation.

Description

ReadHardId retrieves an ID value (1 to 126) from either an add-on hardware ID device on an X-Bus workstation or the non-volatile RAM on an X-Bus or X-Bus+ workstation. The value of the hardware ID is saved even if the workstation is powered off.

On X-Bus workstations, if there is no physical hardware ID device, status code 693 ("No device present on the I-bus") is returned. On SuperGen and 286i/386i workstations, if a loss of power has corrupted the memory where the ID number is stored, status code 696 ("No data available") is returned.

Procedural Interface

ReadHardId (pId): ercType

where

pId

is the memory address of a word to which the hardware ID is returned.

Request Block

sId is always 2.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	2
1	RtCode	1	0
2	nReqPbCb	1	1
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	8192
12	reserved	2	
14	pId	4	
18	sId	2	2

ReadInputEvent (wMode, pEventBlock, sEventBlock): ercType

Description

ReadInputEvent returns one input event from an input device. ReadInputEvent does not Read from a submit file. ReadInputEvent currently returns one of the following event types:

- pressing a keyboard key
- pressing a mouse button
- moving out of a motion rectangle

If the user presses a mouse button and moves the mouse at the same time, the mouse button event is returned first, then the motion rectangle event.

Because ReadInputEvent returns both keyboard and mouse events, the ReadKbdDirect operation is not necessary and should not be used at the same time as ReadInputEvent. As in ReadKbdDirect, special modes allow testing for input in the type-ahead buffer. This interface also provides for any of the other defined-input events in addition to keystrokes (pressing a mouse button; moving out of a motion rectangle).

When a keyboard event occurs, the first call to ReadInputEvent returns the raw unencoded key value.

See also ReadInputEventNSC, which uses normalized screen coordinates.

Procedural Interface

ReadInputEvent (wMode, pEventBlock, sEventBlock): ercType

where

wMode

corresponds to the same modes used in ReadKbdDirect. (See the description of ReadKbdDirect.) In unencoded mode, a keyboard code is used in place of a character code. The four values of *wMode* are

Value	**Description**
0	Wait until a character is typed, then return it.
1	If a character code is currently available, return it. If no character code is available, return status code 602.
2	Wait until a character code is available. Then return a copy of it, but do not remove it from the type-ahead buffer.
3	If a character code is available, return a copy of it, but do not remove it from the type-ahead buffer. If no character code is available, return status code 602.

pEventBlock

is the memory address of an event block structure that defines the event. The format of the event block structure is shown below:

Offset	Field	Size (bytes)	Description
0	wEvent	2	Input event type is returned here. Event types currently implemented are: 20h = Keystroke. 21h = Repeat keystroke. 80h = Mouse button pressed. 60h = Mouse cursor moved out of motion rectangle.
2	bCodeRet	1	Event code is returned here. For a keystroke, the event code is a character code or a keyboard code in unencoded mode. For a mouse, the event code is a button code. The low order 3 bits of bCodeRet are used for the button code. The interpretation of bits 0 through 2 is as follows:

Bit	Value	Attribute
0	1	Right
1	2	Middle
2	4	Left

Offset	Field	Size (bytes)	Description
3	wX	2	The horizontal coordinate of the cursor when the event occurred. wX and wY (below) apply only to mouse button and motion rectangle events. The coordinates are specified as a virtual screen coordinates.
5	wY	2	The vertical coordinate of the cursor when the event occurred.
7	mouseTime	8	The amount of time since the mouse button was last pressed.
15	wStatus	2	Described below.
17	wStatus1	2	A status word. If this value is 4, the chord that resulted from emulation was produced by a chord which is not supported.
19	keyboardCode	1	The emulated or raw keyboard code. If this field contains a raw keyboard code, the *rawKey* bit in *wStatus* is set to 1.

Offset	Field	Size (bytes)	Description
20	sChar	1	The size of the character code. Each value indicates the following:
			Value / **Description**
			0 / 8 bits
			1 / 16 bits
			2 / 32 bits
21	chordState	2	The chord state. For the default K1 keyboard data block, each set bit number indicates the following:
			Bit Number / **Chord**
			0 / Left **Shift**
			1 / Right **Shift**
			2 / Left **Code**
			3 / Right **Code**
			4 / **Lock**
			5-15 / Unused
23	characterCode	4	The character code.

In the *wStatus* field, each bit position, when set, indicates the following:

Field	**Bit Number**	**Description**
diacritResult	0	The character is the result of a diacritical translation.
afterDiacrit	1	The character follows a diacritical key.
beginDiacrit	2	The key may begin a diacritical pair.
string	3	The character translates to a string.
nullVal	4	No character code is defined for that key (for example, **Shift**).
transVal	5	A character code is defined for the key.
chord	6	The key is a chord.
upstroke	7	The keyboard event is an upstroke.
repeatChar	8	The character was produced because a key was held down.
repeatAttr	9	The character will continue to repeat if this key is held down.
noMatch	10	The character generated by this key follows a diacritical start character and did not match.
UserSpecial	11	In the Repeat Attributes Table, the command bit (bit 2) is set in the entry for the key.

Field	Bit Number	Description
haveChar	12	Set by Asian keyboard translation procedures to indicate that there is more than one byte of character data to be returned.
asiaMore	13	Set by Asian keyboard translation procedures.
command	14	The key is defined in the Command Masks Table as a command key. (See the Command Masks Table in Chapter 4.)
rawKey	15	The returned value is a raw keyboard code. In this case, no other bits in the *wStatus* field are set.

sEventBlock

the size of the event block in bytes. To retrieve all information, set *sEventBlock* to 27.

Request Block

ReadInputEvent is an object module procedure in Mouse.lib.

This page intentionally left blank.

ReadInputEventNSC (wMode, pEventBlock, sEventBlock): ercType

Description

ReadInputEventNSC returns one input event from an input device; it does not Read from a submit file. ReadInputEventNSC currently returns one of three event types: pressing a keyboard key, pressing a mouse button, or moving out of a motion rectangle.

If the user presses a mouse button and moves the mouse at the same time, the mouse button event is returned first, then the motion rectangle event.

Because ReadInputEventNSC returns both keyboard and mouse events, the ReadKbdDirect operation is not necessary and should not be used at the same time as ReadInputEventNSC.

Special modes allow testing for input in the type-ahead buffer. This interface also provides for any of the other defined-input events in addition to keystrokes (pressing a mouse button; moving out of a motion rectangle).

Procedural Interface

ReadInputEventNSC (wMode, pEventBlock, sEventBlock): ercType

where

wMode

corresponds to the same modes used in ReadInputEvent. (For the values of *wMode*, see the description of ReadInputEvent.)

pEventBlock

is the memory address of a structure that defines the event. (For a detailed description of the event block structure, see ReadInputEvent. The only difference is that values for *wX* and *wY* are normalized screen coordinates.)

sEventBlock

is the size of the event block in bytes. To retrieve all information, set *sEventBlock* to 27.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntlInfo	1	6
1	rtInfo	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	1
4	UserNum	2	
6	exchRet	2	
8	ercRet	2	
10	rqCode	2	32791
12	wMode	2	
14	reserved	4	
18	pEventBlock	4	
22	sEventBlock	2	

ReadKbd (pCharRet): ercType

New applications should use ReadKbdInfo instead of ReadKbd. (See "Keyboard and I-Bus Management" in the CTOS Operating System Concepts Manual *for details.*

Description

ReadKbd reads one character from the keyboard. If a submit file is currently active, ReadKbd reads the character from that file instead of from the keyboard.

Procedural Interface

ReadKbd (pCharRet): ercType

where

pCharRet

is the memory address of a byte to which the character is to be returned.

mode

is 0 by default. (The values of *mode* are described in the ReadKbdDirect operation.) If, however, you build the request block (rather than using the procedural interface), you can provide a mode other than 0 at offset 12.

Request Block

sCharRet is always 2.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	53
12	mode	2	
14	reserved	4	
18	pCharRet	4	
22	sCharRet	2	2

ReadKbdDataDirect (mode, pInfoRet, sInfoRet): ercType

NOTE: An application should use ReadKbdInfo in place of ReadKbdDataDirect. Unlike ReadKbdDataDirect, ReadKbdInfo can retrieve both single-byte and multibyte characters.

Description

ReadKbdDataDirect reads and returns one character code (or keyboard code, if in unencoded mode) from the keyboard. In addition, ReadKbdDataDirect returns information about the keyboard state. ReadKbdDataDirect never reads from a submit file. Special modes permit testing for the presence of a character in the type–ahead buffer.

ReadKbdDataDirect is the same as ReadKbdDirect in all of the ways described above except that ReadKbdDirect does not return keyboard state information.

In many cases, ReadKbdDataDirect eliminates the need to read the keyboard in unencoded mode, because all of the information required to yield the character code is returned to the caller in this operation.

Procedural Interface

ReadKbdDataDirect (mode, pInfoRet, sInfoRet): ercType where

mode

is one of the following codes:

Code	**Description**
0	Wait until a character code (or a keyboard code, if in unencoded mode) is available, then return it.
1	If a character code (or a keyboard code, if in unencoded mode) is currently available, return it. If no character code or keyboard code is available, return status code 602 ("No character available").
2	Wait until a character code (or a keyboard code, if in unencoded mode) is available, then return a copy of it but do *not* remove it from the type-ahead buffer. A subsequent call to ReadKbdDirect or ReadKbd reads the same character code or keyboard code again.
3	If a character code (or a keyboard code, if in unencoded mode) is available, return a copy of it but do *not* remove it from the type-ahead buffer. If no character code or keyboard code is available, return status code 602 ("No character available").

pInfoRet

is the memory address of a 10 byte structure into which information about the keyboard event is returned. The format of the structure follows:

Offset	Field	Size (Bytes)	Description
0	iKey	1	the keyboard code
1	rsvd	1	reserved
2	bAttr*	1	byte describing the keyboard code attributes: bit 0: diacritical key when shifted bit 1: diacritical key when not shifted bit 2: reserved bit 3-4: repeat frequency 00 = 1 per second 01 = 10 per second 10 = 20 per second 11 = 30 per second bit 5-6: initial delay (in milliseconds) before repeating 00 = no repeating 01 = 200 ms 10 = 400 ms 11 = 700 ms

*One byte of the 3-byte entry in the Keyboard Encoding table used to map the keyboard event to a character code.

Offset	Field	Size (Bytes)	Description
3	bLower*	1	the character code if unshifted (in CTOS II 3.3 and later, this is the value in table 0).
4	bUpper*	1	the character code if shifted (in CTOS II 3.3 and later, this is the value in table 1).
5	rsvd	1	reserved
6	bShift	1	byte describing the keyboard shift state:
			bit 0: left Shift key depressed
			bit 1: right Shift key depressed
			bit 2: left Code key depressed
			bit 3: right Code key depressed
			bit 4: Shift Lock down
			bit 5-7: reserved
7	bChar	1	the character code based on the current shift state
8	rsvd	2	reserved

*One byte of the 3-byte entry in the Keyboard Encoding table used to map the keyboard event to a character code.

sInfoRet

is the maximum size of the data to be returned.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	347
12	mode	2	
14	reserved	4	
18	pInfoRet	4	
22	sInfoRet	2	

This page intentionally left blank

ReadKbdDirect (mode, pCharRet): ercType

New applications should use ReadKbdInfo instead of ReadKbdDirect. (See "Keyboard and I-Bus Management" in the CTOS Operating System Concepts Manual *for details.*

Description

ReadKbdDirect reads and returns one character code (or keyboard code, if in unencoded mode) from the keyboard. ReadKbdDirect never reads from a submit file. Special modes permit testing for the presence of a character in the type-ahead buffer.

Procedural Interface

ReadKbdDirect (mode, pCharRet): ercType

where

mode

is one of the following codes:

Code	**Description**
0	Wait until a character code (or keyboard code, if in unencoded mode) is available, then return it.
1	If a character code (or keyboard code, if in unencoded mode) is currently available, return it. If no character code or keyboard code is available, return status code 602 ("No character available").

Code	**Description**
2	Wait until a character code (or keyboard code, if in unencoded mode) is available, then return a copy of the code but do *not* remove it from the type-ahead buffer. A subsequent ReadKbdDirect or ReadKbd operation reads the same character code or keyboard code again.
3	If a character code (or keyboard code, if in unencoded mode) is available, return a copy of it but do *not* remove it from the type-ahead buffer. If no character code or keyboard code is available, return status code 602 ("No character available").

pCharRet

is the memory address of a byte to which a character code or keyboard code is returned.

Request Block

sCharRet is always 1.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	54
12	mode	2	
14	reserved	4	
18	pCharRet	4	
22	sCharRet	2	1

This page intentionally left blank

ReadKbdInfo (mode, pRecordRet, sRecordRet): ercType

Description

ReadKbdInfo returns a detailed description of a keyboard event in a structure called the Keyboard Record. Information provided by ReadKbdInfo includes the emulated keyboard code, the character code (if one is defined for that key), the size of the character code, and the status of the keyboard event.

Using ReadKbdInfo, an application can run in character mode yet still obtain detailed information about a keyboard event. In addition, ReadKbdInfo can return either a single-byte character or a multibyte character. For these reasons, most new applications should use ReadKbdInfo to obtain keyboard input. (See "Keyboard and I-Bus Management" in the *CTOS Operating System Concepts Manual* for details.)

If the application is running in unencoded mode, ReadKbdInfo returns keystroke data after an emulated keyboard code is produced. For character mode applications, ReadKbdInfo optionally returns keystroke data after each keyboard event. (See SetKbdUnencodedMode for details on setting the keyboard mode.)

If the application is running in either unencoded plus mode or character plus mode, ReadKbdInfo returns a raw keyboard code and sets the *rawKey* bit in the *wStatus* field of the Keyboard Record. When the application calls ReadKbdInfo a second time, the emulated keyboard code is returned.

When bit 5 of the *mode* parameter is cleared and the application is running in character mode, ReadKbdInfo reads from a submit file if one is currently active.

Status code 616 ("Bad keyboard ID") is returned if the emulation data block of the application is not valid for the attached keyboard. Status code 617 ("Bad keyboard data block") is returned if the data block of the application is damaged.

Procedural Interface

ReadKbdInfo (mode, pRecordRet, sRecordRet): ercType

where

mode

specifies the mode of the read operation. Each bit, when set or cleared, causes ReadKbdInfo to perform the following function:

Bit Number	**Value**	**Description**
0 (block/no block)	0	Wait until keyboard data is available and then return a keyboard record. (See the *pRecordRet* parameter.)
	1	If keyboard data is available, return a keyboard record. Otherwise, return status code 602 ("No character available").
1 (peek/read)	0	Remove the contents of the type-ahead buffer.
	1	Do not remove the contents of the type-ahead buffer.
4 (no filter/filter)	0	Return keystroke data only when a downstroke, or series of downstrokes, produces a character code. For unencoded applications, the ReadKbdInfo operation always returns after an emulated keyboard code is produced, regardless of the value of bit 4. For applications in either unencoded plus mode or character plus mode, ReadKbdInfo always returns after a raw keyboard code is produced, regardless of the value of bit 4.

Bit Number	Value	Description
	1	Return keystroke data after any keyboard event (for example, an upstroke or a downstroke).
5 (file/keyboard)	0	Return keystroke data from a submit file if one is currently active.
	1	Return keystroke data from the keyboard even if a submit file is active.

pRecordRet

is the memory address of an area to which the Keyboard Record is to be returned. This record has the following format:

Offset	Field	Size (Bytes)	Description
0	wStatus	2	Described below.
2	wStatus1	2	A status word. If this value is 4, the chord that resulted from emulation was produced by a chord which is not supported.
4	kbdCode	1	The emulated or raw keyboard code. If this field contains a raw keyboard code, the *rawKey* bit in *wStatus* is set to 1, and the other fields in the Keyboard Record are set to 0.
5	sChar	1	The size, in bytes, of the character code.
6	chordState	2	The chord state. Each bit number, when set, indicates a chord is active. The default values that occur if the released version of NlsKdb.sys are the following:

Bit Number	Chord
0	Left **Shift**
1	Right **Shift**

Bit Number	**Chord**
2	Left **Code**
3	Right **Code**
4	**Lock**
5	**Alt** (K3, K5, K6) Left **Alt** (SG102-K)
6	Right **Alt** (SG102-K)
7	**Num Lock** (SG102-K)
8-15	Unused

8	characterCode	4	The character code.

Note that ReadKbdInfo only updates the *sChar* and *transChar* fields if a character code is defined for the keystroke. In the *wStatus* field, each set bit position indicates the following:

Field	**Bit Number**	**Description**
diacritResult	0	The character is the result of a diacritical translation.
afterDiacrit	1	The character follows a diacritical key.
beginDiacrit	2	The key may begin a diacritcal pair.
string	3	The character translates to a string.
nullVal	4	No character code is defined for that key (for example, **Shift**).
transVal	5	A character code is defined for the key.

Field	**Bit Number**	**Description**
chord	6	The key is a chord.
upstroke	7	The keyboard event is an upstroke.
repeatChar	8	The character was produced because a key was held down.
repeatAttr	9	The character will continue to repeat if this key is held down.
noMatch	10	The character generated by this key follows a diacritical start character and did not match.
user	11	In the Repeat Attributes Table, the command bit (bit 2) is set in the entry for the key.
haveChar	13	Set by Asian keyboard translation procedures to indicate that there is more than one byte of character data to be returned.
asiaMore	12	Set by Asian keyboard translation procedures.
command	14	The key is defined in the Command Masks Table as a command key. (See the Command Masks Table in Chapter 4.)
rawKey	15	The returned value is a raw keyboard code. In this case, no other bits in the *wStatus* field are set.

sRecordRet

is the size of the memory area where the Keyboard Record is to be returned.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	403
12	mode	2	
14	reserved	4	
18	pRecordRet	4	
22	sRecordRet	2	

ReadKbdStatus (wSequence, pUserNumRet, pStatusRet): ercType

Description

ReadKbdStatus returns the keyboard status under the following conditions:

- when the keyboard state changes for user numbers other than the current keyboard owner
- during that part of a context switch when the partition–managing program calls AssignKbd and switches the user number owning the keyboard to a different user number

For either condition above, the status flag returned is TRUE if the user number still has an outstanding ReadKbd request because no characters were available to be read. For the second condition above, the flag is FALSE if the keyboard owner changes and the prior keyboard owner had no outstanding ReadKbd requests.

ReadKbdStatus is used by partition–managing programs such as the Context Manager, for example, to reflect the current keyboard state on its status screen.

Status code 611 ("Issue ReadKbdStatus more often") is returned if ReadKbdStatus is called too infrequently to capture all status changes. To avoid this status code, programs calling ReadKbdStatus should build a request block and call request directly. (For details on making direct requests, see "Interprocess Communication" in the *CTOS Operating System Concepts Manual.*)

Procedural Interface

ReadKbdStatus (wSequence, pUserNumRet, pStatusRet): ercType

where

wSequence

is initially 0. Thereafter, this field is managed by the Keyboard process.

pUserNumRet

is memory address of the user number whose keyboard status is being read.

pStatusRet

is the memory address of the user number's status flag.

Request Block

sUserNumRet is always 2 and *sStatusRet* is always 1.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	2
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	257
12	wSequence	2	
14	reserved	4	
18	pUserNumRet	4	
22	sUserNumRet	2	2
24	pStatusRet	4	
28	sStatusRet	2	1

This page intentionally left blank

ReadKeyedQueueEntry (pbQueueName, cbQueueName, pbKey1, cbKey1, oKey1, pbKey2, cbKey2, oKey2, pEntryRet, sEntryRet, pStatusBlock, sStatusBlock): ercType

Description

ReadKeyedQueueEntry is used by the caller to obtain the first queue entry with up to two key fields equal to the values specified. ReadKeyedQueueEntry reads the entry into a buffer and returns the Queue Status Block.

The byte count of at least one key field (either *cbKey1* or *cbKey2*) must be nonzero. If only one is nonzero, only that key field is used in the search. If both are nonzero, both are used in the search.

Each nonzero key field must match a specified sb string in the queue entry. In an sb string, the first byte contains the byte count of the string in binary.

Procedural Interface

ReadKeyedQueueEntry (pbQueueName, cbQueueName, pbKey1, cbKey1, oKey1, pbKey2, cbKey2, oKey2, pEntryRet, sEntryRet, pStatusBlock, sStatusBlock): ercType

where

pbQueueName
cbQueueName

describe a queue name corresponding to a queue name specified when the queue was installed.

pbKey1
cbKey1

describe a key field to be compared with an sb string located at an offset *oKey1* in the queue entry.

oKey1

is the offset of the sb string key field in the queue entry. The offset starts from byte 40 (the first byte of the type-specific portion of the queue entry).

pbKey2
cbKey2

describe a second key field to be compared with an sb string located at an offset *oKey2* in the queue entry.

oKey2

is the offset of the second sb string key field in the queue entry. The offset starts from byte 40 (the first byte of the type-specific portion of the queue entry).

pEntryRet
sEntryRet

describe the buffer where the queue entry is read.

pStatusBlock
sStatusBlock

describe the buffer where the Queue Status Block for the queue entry is returned.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	4
1	RtCode	1	0
2	nReqPbCb	3	
3	nRespPbCb	2	
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	140
12	oKey1	2	
14	oKey2	2	
16	pbQueueName	4	
20	cbQueueName	2	
22	pbKey1	4	
26	cbKey1	2	
28	pbKey2	4	
32	cbKey2	2	
34	pEntryRet	4	
38	sEntryRet	2	
40	pStatusBlock	4	
44	sStatusBlock	2	

This page intentionally left blank

ReadKeySwitch: word

NOTE: This operation takes no parameters and is only used on shared resource processors. Otherwise, a value of 0 is returned.

Description

ReadKeySwitch returns a word value that indicates the keyswitch position in which a shared resource processor was booted. The word returned is one of the following values:

Value	**Meaning**
0	Execution is not on a shared resource processor
1	Manual keyswitch position
2	Remote keyswitch position
3	Normal keyswitch position

Procedural Interface

ReadKeySwitch: word

ReadKeySwitch takes no parameters.

Request Block

ReadKeySwitch is an object module procedure.

This page intentionally left blank

ReadMcr (mode, pBufferRet, sBufferMax, pcbRet): ercType

Description

ReadMcr reads the magnetic card reader (MCR) data according to various operation modes.

Note that if the caller does not select one of the operation modes to clear the MCR buffer, either a call to PurgeMcr or another call to ReadMcr must be made with a mode that clears the buffer. The buffer will automatically be purged by the operating system after an elapsed period of time or on the next swipe of the MCR.

Procedural Interface

ReadMcr (mode, pBufferRet, sBufferMax, pcbRet): ercType

where

mode

is a word value that specifies the operation mode. The values of *mode* are

Value	**Description**
0	Wait until MCR data is available, return the data, and clear the MCR buffer.
1	If MCR data is available, return the data and clear the MCR buffer. Otherwise return status code 696 ("Data not available").
2	Wait until MCR data is available, return the data but do not clear the MCR buffer.
3	If MCR data is available, return the data but do not clear the MCR buffer. Otherwise return status code 696 ("Data not available").

pBufferRet

is the memory address of the buffer.

sBufferMax

is the size (word) of the buffer to which the MCR data is returned. If more data is returned than can fit into the buffer, the data is truncated.

pcbRet

is the memory address of a word that contains the count of bytes returned.

Request Block

scbRet is always 2.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	2	6
2	nReqPbCb	1	0
3	nRespPbCb	1	2
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	8234*
12	mode	2	
14	reserved	4	0
18	pBufferRet	4	
22	sBufferMax	2	
24	pcbRet	4	
28	scbRet	2	2

*On operating systems prior to CTOS I 3.3, the request code is 65523 or −13.

This page intentionally left blank

ReadNextQueueEntry (pbQueueName, cbQueueName, qeh, pEntryRet, sEntryRet, pStatusBlock, sStatusBlock): ercType

Description

ReadNextQueueEntry is used by the caller to obtain a list of queue entries. ReadNextQueueEntry reads an entry from the specified queue into a buffer and returns the Queue Status Block, which contains the queue entry handle of the next entry in the queue. The entry data returned starts at byte 40 (first byte of the type-specific portion) of the first queue entry sector.

If another caller removes the next queue entry before it is read, status code 904 ("Entry deleted") is returned on any attempt to read that entry.

Procedural Interface

ReadNextQueueEntry (pbQueueName, cbQueueName, qeh, pEntryRet, sEntryRet, pStatusBlock, sStatusBlock): ercType

where

pbQueueName
cbQueueName

describe a queue name corresponding to a queue name specified when the queue was installed.

qeh

is the 32-bit queue entry handle returned from a previous call to the MarkKeyedQueueEntry or MarkNextQueueEntry operation. A 0 indicates the first entry in the queue.

pEntryRet
sEntryRet

describe the buffer where the queue entry is read.

pStatusBlock
sStatusBlock

describe the buffer where the Queue Status Block for the queue entry is returned.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	4
1	RtCode	1	0
2	nReqPbCb	1	1
3	nRespPbCb	1	2
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	139
12	qeh	4	
16	pbQueueName	4	
20	cbQueueName	2	
22	pEntryRet	4	
26	sEntryRet	2	
28	pStatusBlock	4	
32	sStatusBlock	2	

ReadOsKbdTable (mode, pDataBlock, sDataBlock): ercType

Description

ReadOsKbdTable copies the operating system translation or emulation data block to the specified memory location. This operation enables the caller to examine and change its own copy of the operating system data block. To post the modified data block, the caller can then use PostKbdTable. (See PostKbdTable.)

The calling program must specify the memory area to which the operating system data block is to be copied. If the caller specifies a size of 0 for this memory area, ReadOsKbdTable only returns the size of the data block at *pDataBlock*. Using this method, the caller can determine the correct amount of memory to allocate for the data block.

Status code 617 ("Bad keyboard data block") is returned if inadequate space exists where the data block is to be copied. This status code is also returned if the caller tries to copy an emulation data block when one does not exist.

Procedural Interface

ReadOsKbdTable (mode, pDataBlock, sDataBlock): ercType where

mode

specifes the type of data block to be copied. Each value indicates the following:

Value	**Description**
0	Copy the operating system's Translation Data Block.
1	Copy the operating system's Emulation Data Block.

If the caller specifies a value of 1, and no emulation data block is installed, status code 617 ("Bad keyboard data block") is returned.

pDataBlock
sDataBlock

describe the buffer to which the keyboard data block is to be copied. If *sDataBlock* is 0 or smaller than the specified data block, the data block size is returned to the buffer at the memory address *pDataBlock*. In addition, status code 617 ("Bad keyboard data block") is returned.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	1
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	399
12	mode	2	
14	reserved	4	
18	pDataBlock	4	
22	sDataBlock	2	

This page intentionally left blank

ReadPSLog (sh, pbLogData, cbLogData, psDataRet): ercType

Description

ReadPSLog is a synonym for PSReadLog. See the description of PSReadLog for details.

This page intentionally left blank

ReadRemote has no procedural interface. You must make a request block and issue the request using Request, RequestDirect, or RequestRemote.

Caution: *This operation is for* **restricted use only** *and is subject to change in future operating system releases. It works only on shared resource processor hardware.*

Description

ReadRemote allows the caller to read from disk directly to the memory of any processor board on a shared resource processor.

Procedural Interface

ReadRemote has no procedural interface. You must make a request block and issue the request using one of the request operations listed above

where

fh

is a file handle returned from an OpenFile operation. The device can be open in either read or modify mode.

baBuffer

is the bus address of the first byte of the buffer to which the data is to be read. The buffer must be word aligned.

sBuffer

is the count of bytes to be read to memory. It must be a multiple of the sector size (128, 256, 512, or 1024).

lfa

is the byte offset, from the beginning of the file, of the first byte to be read. It must be a multiple of the sector size (128, 256, 512, or 1024).

psDataRet

is the memory address of the word to which the count of bytes successfully read is to be returned.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	slotNumber*	1	0
2	nReqPbCb	1	1
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	12325
12	fh	2	
14	lfa	4	
18	baBuffer	4	
22	sBuffer	2	
24	psDataRet	4	
28	ssDataRet	2	2

*The slot number of the board to which the data is to be read

ReadRsRecord (pRswa, pRecordRet, sRecordMax, pcbRet): ercType

Description

ReadRsRecord reads the next record from the open RSAM file identified by the memory address of the Record Sequential Work Area.

Procedural Interface

ReadRsRecord (pRswa, pRecordRet, sRecordMax, pcbRet): ercType

where

pRswa

is the memory address of the same Record Sequential Work Area that was supplied to OpenRsFile.

pRecordRet
sRecordMax

describe the memory area to which the record is to be read.

pcbRet

is the memory address of the word to which the number of bytes read is returned. If the record fits in the supplied memory area, *pcbRet* is the length of the record. If the record does not fit in the supplied memory area, *pcbRet* is *sRecordMax* and status code 3606 ("Record too large") is returned.

Request Block

ReadRsRecord is an object module procedure.

This page intentionally left blank

ReadStatusC (pBswa, wStatusMask, pwStatusRet): ercType

Description

ReadStatusC reads the values of the specified status bits. *wStatusMask* is a word containing the selected mask bits from the list below (read only status lines):

Value	**Description**
1	CD(Carrier Detect)
2	CTS(Clear To Send)
4	SRX(Secondary Receive)
8	DSR(Data Set Ready)
16	RNG(Ring Indicator)

Masks may be ORed together to reference more than one status bit at a time. Only the status bits selected by *wStatusMask* are accessed. The others are not read. (For more information on the communications programming interfaces, see "Communications Programming" in the *CTOS Operating System Concepts Manual.*)

Procedural Interface

ReadStatusC (pBswa, wStatusMask, pwStatusRet): ercType

where

pBswa

is the memory address of the Byte Stream Work Area (BSWA).

wStatusMask

is a word containing the selected mask bits described above.

pwStatusRet

is the memory address of a word to which the status indicators, corresponding bit-for-bit with the selected mask bits, are returned.

Request Block

ReadStatusC is an object module procedure.

ReadTerminal has no procedural interface. You must make a request block and issue the request using Request, RequestDirect, or RequestRemote.

Caution: *This operation works on the shared resource processor TP and CP boards only. Do not use it on workstation hardware.*

Description

This request is used by the client to read data from one of the asynchronous ports on a Cluster Processor or Terminal Processor.

The user specifies a buffer size and a timeout value. ReadTerminal returns when one of the following three conditions occurs:

- The number of requested characters is received.
- The timeout has elapsed. The timeout begins after the first character is received.
- An error is detected.

Procedural Interface

ReadTerminal has no procedural interface. You must make a request block and issue the request using one of the request operations listed above.

where

bDestCpu

is the slot number of the Cluster or Terminal Processors connected to an RS-232-C port .

bSourceCpu

is the slot number of the client.

bPort

is the port number of a terminal attached to an RS-232-C port.

bWindow

must be zero.

bTime

is the number of 20-millisecond time units to wait for the request to be satisfied after the first character is received. This allows a large amount of data to be received, particularly if the data rate is high. If there is little data, the timeout causes a partially filled buffer to be returned.

bDummy

is a filler byte that ensures word alignment of the rest of the request block.

psCountRet
ssCountRet

describe the area to receive the count of the number of bytes actually read. The returned count will be in linear format (byte swapped from normal 80186 usage).

pDataRet
sDataMax

describe the area to receive the output data.

Request Block

ssCountRet **is always 2.**

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	2
4	userNum	2	
6	ExchResp	2	
8	ercRet	2	
10	rqCode	2	12309
12	bDestCPU	1	
13	bSourceCPU	1	
14	bPort	1	
15	bWindow	1	
16	bTime	1	
17	bDummy	1	
18	psCountRet	4	
22	ssCountRet	2	2
24	pDataRet	4	
28	sDataMax	2	

ReadToNextField (fh, pBuffer, sBuffer, pLines, cLinesMax, wLineSize, pcLines): ercType

Description

ReadToNextField reads from a file and searches for a ':FieldName:' string. The search begins at the current location in the file. The text strings between the beginning file location and the ':Fieldname:' string are stored in a line array.

At the end of the operation, the current file location is set to the beginning of the ':Fieldname:' string. A subsequent LookUpField can the read the ':Fieldname:' string.

Procedural Interface

ReadToNextField (fh, pBuffer, sBuffer, pLines, cLinesMax, wLineSize, pcLines): ercType

where

fh

is the file handle of the file to be searched.

pBuffer
sBuffer

describe a buffer to be used in the search. *sBuffer* must be a multiple of 512.

pLines

is the memory address of a buffer where the string(s) read are returned.

cLinesMax

is the maximum number of lines to read.

wLineSize

determines mode of storage. If *wLinesSize* is zero, the strings are stored back to back as (*cb*, *string*)(*cb*, *string*), where the strings are variable size. If wLinesize is greater than zero, the strings are stored as fixed-length entities.

pcLines

is the memory address of where the count of lines read is returned.

Request Block

ReadToNextField is an object module procedure.

ReallocHugeMemory (sn, cbResize): ercType

Caution: *This operation only works on virtual memory operating systems and is for use by programs executing in protected mode.*

Description

ReallocHugeMemory resizes either memory allocated by the AllocHugeMemory operation or an expand up data segment of size less than 64K bytes that is owned by the caller.

In the first case, the maximum new size is limited to the value supplied in the AllocHugeMemory operation. In the second case, the maximum new size is limited to 64K bytes.

Procedural Interface

ReallocHugeMemory (sn, cbResize): ercType

where

sn

is the first selector (word) that addresses the memory to be resized.

cbResize

is the desired memory size (dword).

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	463
12	sn	2	
14	cbResize	4	

ReceiveCommLineDma (commLineHandle, pBufferRet, sBufferMax, psDataRet): ercType

Description

ReceiveCommLineDma sets up the DMA controller to transfer a specified number of bytes from the communications channel to the specified memory buffer.

This operation returns immediately after the DMA controller has been set up, rather than waiting for the specified number of bytes to be received. The count of bytes received is from the previous ReceiveCommLineDma call. An application also may query this byte count by using the GetCommLineDmaStatus operation.

Because ReceiveCommLineDma is a system-common procedure, it may be called from a communications interrupt handler.

Procedural Interface

ReceiveCommLineDma (commLineHandle, pBufferRet, sBufferMax, psDataRet): ercType

where

commLineHandle

is a handle returned by InitCommLine.

pBufferRet

is the memory address of the first byte of the buffer where the data is to be read.

sBufferMax

is the maximum number of bytes to be read to memory.

psDataRet

is the memory address of the word where the count of bytes successfully read by the previous ReceiveCommLineDma request is placed.

Request Block

ReceiveCommLineDma is a system−common procedure.

ReinitLargeOverlays (cParasSwapBuffer)

Description

ReinitLargeOverlays allows you to change the size of the overlay buffer in a program using the Virtual Code facility to recover memory or to extend the overlay buffer for better performance. The overlay buffer size can be changed only by adding or subtracting memory from the high memory side of the buffer.

Use this operation if the overlay zone is only one of several entities competing for memory space. For example, if a growing heap is also present, you may wish to resize the overlay zone.

ReinitLargeOverlays is identical to ReinitOverlays with the following exception: you would use ReinitLargeOverlays if you described the overlay buffer as a count of 16-byte paragraphs instead of as a count of bytes. Using ReinitLargeOverlays, you can have a buffer that is up to 1M–byte in size.

Procedural Interface

ReinitLargeOverlays (cParasSwapBuffer)

where

cParasSwapBuffer

is the size of the overlay buffer in 16–byte paragraphs. The buffer must be large enough to contain the largest nonresident code segment.

Request Block

ReinitLargeOverlays is an object module procedure.

ReinitOverlays (sSwapBuffer): ercType

Description

ReinitOverlays allows you to change the size of the overlay buffer in a program that uses the Virtual Code facility to recover memory or to extend the overlay buffer for better performance.

Use of ReinitOverlays is identical to ReinitLargeOverlays with the following exception: use ReinitOverlays if you described the overlay buffer as a count of bytes instead of as a count of 16-byte paragraphs. (See ReinitLargeOverlays.)

Procedural Interface

ReinitOverlays (sSwapBuffer): ercType

where

sSwapBuffer

is the size of the overlay buffer. The buffer must be large enough to contain the largest nonresident code segment.

Request Block

ReinitOverlays is an object module procedure.

This page intentionally left blank

ReinitStubs (pStaticsDesc): ercType

Caution: *In protected mode, this operation returns status code 7 ("Not implemented").*

Description

ReinitStubs sets the memory addresses of all stubs to reflect the address of OverlayFault in a program using the Virtual Code facility.

This is a one-time setting. After a ReinitStubs call, the Overlay Manager continues to examine the use of procedures in overlays and to overwrite stub addresses with the memory addresses of frequently called procedures.

In contrast to ReinitStubs, calling EnableSwapperOptions with an *fFixupStubs* value of FALSE causes every call thereafter to an overlay procedure to pass through OverlayFault. (See EnableSwapperOptions.)

Procedural Interface

ReinitStubs (pStaticsDesc): ercType

where

pStaticsDesc

is the memory address of the data structure *StaticsDesc* in the resident portion of the Virtual Code program. The array of stubs (*rgStubs*) is contained within *StaticsDesc*.

Request Block

ReinitStubs is an object module procedure.

ReleaseByteStream (pBswa): ercType

Description

ReleaseByteStream abnormally closes the device/file associated with the open output byte stream identified by the memory address of the Byte Stream Work Area. ReleaseByteStream, unlike the CloseByteStream operation, does *not* properly write remaining partially full buffers. ReleaseByteStream should *only* be used when a WriteBsRecord, WriteBytes, or CheckpointBs operation fails due to a device error.

Procedural Interface

ReleaseByteStream (pBswa): ercType

where

pBswa

is the memory address of the same Byte Stream Work Area that was supplied to OpenByteStream.

Request Block

ReleaseByteStream is an object module procedure.

This page intentionally left blank

ReleaseByteStreamC (pBswa): ercType

Description

ReleaseByteStreamC reverses the effect of an OpenByteStreamC or AcquireByteStreamC operation, stopping all receive and transmit operations on a serial byte stream and making the serial port available for use by other users again.

Note that to gracefully close a byte stream, CheckPointBsC should be issued before ReleaseByteStreamC, or output may be lost. (The device-independent operation CloseByteStream does this.)

(See "Device–Dependent SAM" and "Communications Programming" in the *CTOS Operating System Concepts Manual* for more information.)

Procedural Interface

ReleaseByteStreamC (pBswa): ercType

where

pBswa

is the memory address of the Byte Stream Work Area (BSWA).

Request Block

ReleaseByteStreamC is an object module procedure.

This page intentionally left blank

ReleasePermanence: ercType

Description

ReleasePermanence releases all overlays from permanent residence in memory in a program using the Virtual Code facility.

Procedural Interface

ReleasePermanence: ercType

Request Block

ReleasePermanence is an object module procedure.

This page intentionally left blank

ReleaseRsFile (pRswa): ercType

Description

ReleaseRsFile abnormally closes the file associated with the open output RSAM file identified by the memory address of the Record Sequential Work Area. ReleaseRsFile, unlike the CloseRsFile operation, does not properly write remaining partially full buffers. ReleaseRsFile should only be used when a WriteRsRecord or CheckpointRsFile operation fails because of a device error.

Procedural Interface

ReleaseRsFile (pRswa): ercType

where

pRswa

is the memory address of the same Record Sequential Work Area that was supplied to OpenRsFile.

Request Block

ReleaseRsFile is an object module procedure.

This page intentionally left blank

RemakeAliasForServer (sgAlias, userNumOwner, pSgAliasRet): ercType

Description

RemakeAliasForServer remakes the alias selector so that the alias selector will persist across system–common return and respond.

Calls to the RemakeAliasForServer operation should match calls to DeallocAliasForServer.

Procedural Interface

RemakeAliasForServer (sgAlias, userNumOwner, pSgAliasRet): ercType

where

sgAlias

specifies the alias selector to be remade.

userNumOwner

is the user number associated with the alias selector.

pSgAliasRet

is the memory address of a 2–byte memory area where the remade alias selector is returned.

Request Block

sSgAliasRet is always 2.

Offset	Field	Size (Bytes)	Contents
0	sCntlInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	336
12	reserved	2	
14	userNumOwner	2	
16	sgAlias	2	
18	pSgAliasRet	4	
22	sSgAliasRet	2	2

RemakeFh has no procedural interface. You must make a request block and issue the request using Request, RequestDirect, or RequestRemote.

Caution: *RemakeFh is supported only by shared resource processors running CTOS/XE 3.0 or later operating system versions, and by workstations running CTOS/VM 2.0 or later operating system versions.*

Description

Given an existing file handle, RemakeFh creates a new file handle that is associated with the user number of the process issuing this request. If a file handle other than the given handle already exists for the user number of the issuer, that handle is returned but in no case is the given handle itself returned.

RemakeFh is useful for system services that may be passed a file handle by a client.

Procedural Interface

RemakeFh has no procedural interface. You must make a request block and issue the request using one of the request operations listed above

where

fh

is the file handle that is to be remade. This file handle does not have to belong to the user number issuing the request.

userNumFh

is the user number of the file handle that is to be remade.

signature

is a value which, when set to 5601h, indicates that the caller has placed a user number in the *userNumFh* field. On CTOS/XE 3.0.1 and later operating system versions, which allow up to 1023 file handle numbers per user number, RemakeFh cannot be used unless *signature* = 5601h.

pFhRet

is the memory address of a word to which a file handle belonging to the user number that issued the request is returned.

Request Block

sFhRet is always 2.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	4110
12	fh	2	
14	userNumFh	2	
16	signature	2	
18	pFhRet	4	
22	sFhRet	2	2

This page intentionally left blank

RemapBusAddress (busAddress, pb, cb, wDmaBoundaryType,
pBusAddressRet, pCbMappedRet): ercType

NOTE: This operation is for use only by programs executing in protected mode.

Description

RemapBusAddress converts a logical address (in SN:RA form) to a bus address and, on some systems, reprograms hardware necessary for the bus address to generate a reference to the appropriate physical address. Either MapBusAddress or ReserveBusAddress must have been used to obtain a bus address before it can be reprogrammed by this operation. RemapBusAddress avoids the system overhead involved in searching for hardware available to be programmed, but is otherwise identical to MapBusAddress.

(For a description of bus addresses, see "Bus Address Management" in the *CTOS Operating System Concepts Manual.*)

Once the bus address is no longer required, the complementary operation UnmapBusAddress should be used to free any hardware resources that were originally allocated by MapBusAddress or ReserveBusAddress.

Procedural Interface

RemapBusAddress (busAddress, pb, cb, wDmaBoundaryType,
pBusAddressRet, pCbMappedRet): ercType

where

busAddress

is a bus Address (double word) previously obtained from the MapBusAddress or ReserveBusAddress operations.

pb
cb

describe an area in memory that is to be referenced by the returned bus address. Both the beginning logical address of the memory and the length of the memory to be referenced must be specified.

wDmaBoundaryType

is a word value that indicates the DMA hardware limitations relevant to the device for which the bus address is requested. Permissible values are

Value	**Description**
0	noDmaBoundary: No DMA boundary needs to be taken into consideration.
1	byteDmaBoundary: DMA boundaries occur at every 64K bytes of physical memory. For example, a DMA transfer could not cross from 1FFFFh to 20000h without a pause to reprogram the DMA hardware.
2	wordDmaBoundary: DMA boundaries occur at every 64K words of physical memory. For example, a DMA transfer could not cross from 3FFFEh to 40000h without a pause to reprogram the DMA hardware.

The processors of a shared resource processor (including the 80386-based GP family) do not have any DMA boundaries. Dual floppy modules and floppy/hard disk modules for workstations have DMA boundaries that occur at 64 kilobyte boundaries in physical memory, workstation hard disk upgrade and expansion modules have DMA boundaries that occur at 64 kiloword boundaries in physical memory while internal SCSI disks in integrated workstations (Series 286i, Series 386i, and B39) and SCSI upgrade modules do not have DMA boundaries.

pBusAddressRet

is the address of a double word to which the calculated bus address is returned.

pCbMappedRet

is the address of a word to which the actual count of bytes that may be transferred to or from the bus address is returned. Because of DMA boundary limitations, this may be less than the count that was requested and may even be 0.

Request Block

RemapBusAddress is a system-common procedure.

This page intentionally left blank.

RemoteBoot has no procedural interface. You must make a request block and issue the request using Request, RequestDirect, or RequestRemote.

Caution: *This operation is for* **restricted use only** *and is subject to change in future operating system releases. It is supported on shared resource processors only.*

Description

RemoteBoot causes another processor board on a shared resource processor to be bootstrapped with a specified system image. This request may only be issued from the master processor.

The specified remote processor must have passed boot ROM checks and be ready to boot. There must be sufficient memory available (according to the remote processor's CPU Description Table (CDT)) to hold the designated system image. (See Chapter 4, "System Structures," for the format of the CDT.)

The target processor's CDT is updated with current configuration data. A doorbell interrupt initiates the execution of the system image. If the target processor does not set the CDT flag *fOsInitialized* within 30 seconds after the doorbell interrupt, status code 153 ("TargetInit") is returned.

Procedural Interface

RemoteBoot has no procedural interface. You must make a request block and issue the request using one of the request operations listed above

where

fh

is the file handle of the system image to be loaded into the designated processor. This system image must have a valid CTOS.run file header. The loaded image must contain a valid CDT linear address in location 1F8h.

bSlot

is the slot number of the target processor.

fReserved

must be set to zero.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	4
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	ExchResp	2	
8	ercRet	2	
10	rqCode	2	12290
12	fh	2	
14	bSlot	1	
15	fReserved	1	0

This page intentionally left blank

RemoveFfsBrackets (pbLevel, cbLevel, ppbToken, pcbToken, lFfsLeftBracket, lFfsRightBracket): word

Description

RemoveFfsBrackets removes the brackets surrounding a token at the specified level (node, volume, directory, or file name) in the file specification. This operation returns the memory addresses of the token (first character past the left bracket) and the token size.

This operation must be called to remove brackets before calling the GetCanonicalNodeAndVol operation.

Procedural Interface

RemoveFfsBrackets (pbLevel, cbLevel, ppbToken, pcbToken, lFfsLeftBracket, lFfsRightBracket): word

where

pbLevel
cbLevel

describe the specified level (token plus brackets).

ppbToken

is a pointer to the memory address at which the token is returned.

pcbToken

is the memory address at which the token size is returned.

lFfsLeftBracket

is the left bracket character code (word).

lFfsSRightBracket

is the right bracket character code (word).

Request Block

RemoveFfsBrackets is an object module procedure.

RemoveKeyedQueueEntry (pbQueueName, cbQueueName, pbKey1, cbKey1, oKey1, pbKey2, cbKey2, oKey2): ercType

Description

RemoveKeyedQueueEntry is used by the caller to locate an unmarked entry in the specified queue with up to two key fields equal to the values specified and to remove the entry from the queue.

The byte count of at least one key field (either *cbKey1* or *cbKey2*) must be nonzero. If one key field is nonzero, that field only is used in the search. If both are nonzero, both are used in the search. Each nonzero key field must match.

Procedural Interface

RemoveKeyedQueueEntry (pbQueueName, cbQueueName, pbKey1, cbKey1, oKey1, pbKey2, cbKey2, oKey2): ercType

where

pbQueueName
cbQueueName

> describe a queue name corresponding to a queue name specified when the queue was installed.

pbKey1
cbKey1

> describe a key field to be compared with a string located at an offset *oKey1* in the queue entry.

oKey1

> is the offset of the string key field in the queue entry. The offset starts from byte 40 (the first byte of the type-specific portion of the queue entry).

pbKey2
cbKey2

describe a second key field to be compared with a string located at an offset *oKey2* in the queue entry.

oKey2

is the offset of the second string key field in the queue entry. The offset starts from byte 40 (the first byte of the type-specific portion of the queue entry).

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	4
1	RtCode	1	0
2	nReqPbCb	1	3
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	138
12	oKey1	2	
14	oKey2	2	
16	pbQueName	4	
20	cbQueName	2	
22	pbKey1	4	
26	cbKey1	2	
28	pbKey2	4	
32	cbKey2	2	

RemoveMarkedQueueEntry (pbQueueName, cbQueueName, qeh): ercType

Description

RemoveMarkedQueueEntry is used by the queue server to remove a previously marked queue entry from the specified queue. The queue entry to be removed is identified by a 32-bit queue entry handle that previously was placed in the Queue Status Block (*qehRet* field) by the MarkKeyedQueueEntry or the MarkNextQueueEntry operation. (See Chapter 4, "System Structures," for the format of the Queue Status Block.)

Procedural Interface

RemoveMarkedQueueEntry (pbQueueName, cbQueueName, qeh): ercType

where

pbQueueName
cbQueueName

describe a queue name corresponding to a queue name specified when the queue was installed.

qeh

is the 32-bit queue entry handle.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	4
1	RtCode	1	0
2	nReqPbCb	1	1
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	143
12	qeh	4	
16	pbQueueName	4	
20	cbQueueName	2	

RemovePartition (userNumPartition): ercType

Description

RemovePartition removes the specified vacant partition.

Procedural Interface

RemovePartition (userNumPartition): ercType

where

userNumPartition

> is the user number returned from a CreatePartition or a GetPartitionHandle operation. Status code 800 ("Partition not vacant") is returned if the partition has not previously been vacated.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	2
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	176
12	userNumPartition	2	

RemoveQueue (qH): ercType

Description

RemoveQueue removes a queue dynamically. As a result of this operation, the specified queue name cannot be specified in subsequent queue operations, because it is no longer recognized by the Queue Manager. The queue entry file associated with this queue name is closed but is not deleted.

Procedural Interface

RemoveQueue (qH): ercType

where

qH

is the queue handle returned by the Queue Manager to the program that installed the queue.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	2
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	32869
12	qH	2	

RenameByteStream (pBswa, pbNewFileSpec, cbNewFileSpec, pbPassword, cbPassword): ercType

Description

RenameByteStream changes the file name and/or the directory name of the file associated with the open byte stream identified by the memory address of the Byte Stream Work Area. Like the **Rename** command, the RenameByteStream operation can rename the file byte stream to another directory on the same volume but cannot rename it to another node or volume. (For details on the **Rename** command, see the *CTOS Executive Reference Manual*.)

Procedural Interface

RenameByteStream (pBswa, pbNewFileSpec, cbNewFileSpec, pbPassword, cbPassword): ercType

where

pBswa

is the memory address of the same Byte Stream Work Area that was supplied to OpenByteStream.

pbNewFileSpec
cbNewFileSpec

describe a character string of the form {Node}[VolName]<DirName>Filename to be used for the new file. The distinction between uppercase and lowercase characters in file specifications is not significant in matching file names.

pbPassword
cbPassword

describe a volume or directory password that authorizes the insertion of a file in the specified directory. It is *not* a password to be assigned to the file being renamed.

Request Block

RenameByteStream is an object module procedure.

RenameFile (fh, pbNewFileSpec, cbNewFileSpec, pbPassword, cbPassword): ercType

Description

RenameFile changes the file name and/or the directory name of an existing file. Like the **Rename** command, the RenameFile operation can rename the file to another directory on the same volume but cannot rename it to another node or volume. (For details on the **Rename** command, see the *CTOS Executive Reference Manual.*)

Procedural Interface

RenameFile (fh, pbNewFileSpec, cbNewFileSpec, pbPassword, cbPassword): ercType

where

fh

is a file handle returned from an OpenFile operation. The file must be open in modify mode.

pbNewFileSpec
cbNewFileSpec

describe a character string of the form {Node}[VolName] <DirName>FileName. The distinction between uppercase and lowercase in file specifications is not significant in matching file names.

pbPassword
cbPassword

describe a volume or directory password that authorizes the insertion of a file in the specified directory. It is *not* a password to be assigned to the file being renamed. The SetFileStatus operation can be used to assign a password to the file being renamed.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	2
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	7
12	fh	2	
14	reserved	4	
18	pbNewFileSpec	4	
22	cbNewFileSpec	2	
24	pbPassword	4	
28	cbPassword	2	

ReopenFile (pFhRet, pbFileSpec, cbFileSpec, pbPassword, cbPassword, mode): ercType

Description

ReopenFile is similar to OpenFile except that if a file handle already exists for that file for the issuing user number, that handle rather than a new one is returned.

ReopenFile is useful for checking if two different file names specify the same file.

Procedural Interface

ReopenFile (pFhRet, pbFileSpec, cbFileSpec, pbPassword, cbPassword, mode): ercType

where

pFhRet

is the memory address of the word where the file handle is returned.

pbFileSpec
cbFileSpec

describe a character string of the form {Node}[Volume]<Directory>FileName. The distinction between uppercase and lowercase in file specifications is not significant in matching file names.

pbPassword
cbPassword

describe a volume or directory password that authorizes access to a file in the specified directory.

mode

is read, modify, or peek. This is indicated by 16-bit values representing the ASCII constants "mr" (mode read), "mm" (mode modify) or "mp" (mode peek). In these ASCII constants, the first character (m) is the high-order byte and the second character is the low-order byte.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	2
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	294
12	reserved	2	
14	mode	2	
16	reserved	4	
18	pbFileSpec	4	
22	cbFileSpec	2	
24	pbPassword	4	
28	cbPassword	2	
30	pFhRet	4	
34	reserved	2	2

Request (pRq): ercType

Description

The Request primitive requests a service by sending a request block to the system service's exchange.

The client can use the Request primitive indirectly simply by using the procedural interface to the system service. However, when it is necessary to overlap execution with the performance of the service, the client can use Request directly.

The Request primitive infers the appropriate service exchange by using the request code of the request block as an index into the Service Exchange Table. The use of the Service Exchange Table allows request codes to remain invariant among operating systems with varying organizations of system service processes. This invariance facilitates the development of filters and is critical to the transparent operation of the cluster configuration.

The client must use AllocExch or QueryDefaultRespExch to acquire an exchange identification to place into the *exchResp* field of the request block.

There must not be conflicting uses of the response exchange specified in the request block; such conflict can cause a malfunction of the application program that is difficult to diagnose.

Procedural Interface

Request (pRq): ercType

where

pRq

is the memory address of the request block.

Request Block

Request is a Kernel primitive.

RequestDirect (exch, prq): ercType

Description

The RequestDirect primitive is used to send a request block to a system service's exchange. Sending the request block is done independently of the default routing implied by the request code in the block. Filter process programs must use RequestDirect and not Send if the system service to which the request block is forwarded is responding to the filter process (two-way pass-through) rather than responding directly to the process issuing the request.

Procedural Interface

RequestDirect (exch, prq): ercType

where

exch

is the exchange to which the request block is to be sent.

prq

is the memory address of the request block.

Request Block

RequestDirect is a Kernel primitive.

This page intentionally left blank

RequestRemote (bRemoteCpuId, pRq): ercType

Caution: *This operation works on shared resource processor hardware only. Do not use it on workstation hardware.*

Description

The RequestRemote primitive requests a system service from a remote SRP processor board by sending the request through Inter-CPU Communication (ICC) to the remote processor. The service exchange to which the request is sent is inferred from the request code in the request block.

This primitive is used when the client expects a response from the remote processor. The eventual response will be routed back to the response exchange specified in the request block without the necessity of special intervention on the part of the system service.

A client may use the RequestRemote primitive directly when it is necessary to bypass the normal request routing mechanisms of the operating system.

NOTE: It is the responsibility of the client to ensure that the specified remote processor actually exists.

Procedural Interface

RequestRemote (bRemoteCpuId, pRq): ercType

where

bRemoteCpuId

is the slot number of the remote processor.

pRq

is the local memory address of the request block.

Request Block

RequestRemote is a Kernel Primitive.

RescheduleMarkedQueueEntry (pbQueueName, cbQueueName, qeh):
ercType

Description

RescheduleMarkedQueueEntry is used by the queue server to remove a previously marked queue entry or to reschedule the entry in the specified queue. Queue entries that specify a repeating time interval are rescheduled; all other entries are removed. The queue entries are identified by a 32–bit queue entry handle that was previously placed in the Queue Status Block (*qehRet* field) by the MarkKeyedQueueEntry or the MarkNextQueueEntry operation. (See Chapter 4, "System Structures," for the format of the Queue Status Block.)

Procedural Interface

RescheduleMarkedQueueEntry (pbQueueName, cbQueueName, qeh):
ercType

where

pbQueueName
cbQueueName

describe a queue name corresponding to a queue name specified when the queue was installed.

qeh

is the 32-bit queue entry handle.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	4
1	RtCode	1	0
2	nReqPbCb	1	1
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	33119
12	qeh	4	
16	pbQueueName	4	
20	cbQueueName	2	

RescheduleProcess (pid): ercType

Description

The RescheduleProcess primitive moves a process in front of all other processes of the same priority on the run queue. RescheduleProcess is useful for implementing time-sliced, round robin scheduling algorithms.

Procedural Interface

RescheduleProcess (pid): ercType

where

pid

> is the process ID (returned by NewProcess or QueryProcessNumber) of the process to be moved in front of other processes of the same priority.

Request Block

RescheduleProcess is a Kernel primitive.

This page intentionally left blank

ReserveBusAddress (cb, pBusAddressRet): ercType

Caution: *This operation is supported by shared resource processor operating systems and is for use by programs executing in protected mode only.*

Description

ReserveBusAddress, on systems that have address mapping hardware, searches for and reserves hardware that may be subsequently programmed by a RemapBusAddress operation. The count of bytes supplied determines what quantity of hardware is reserved. ReserveBusAddress assumes the worst-case for data alignment in subsequent RemapBusAddress operations and always reserves sufficient hardware to accomodate this case. (For a description of bus addresses, see "Bus Address Management" in the *CTOS Operating System Concepts Manual.*)

Once the bus address is no longer required, the complementary operation, UnmapBusAddress, should be used to free any hardware resources that were allocated by ReserveBusAddress.

Procedural Interface

ReserveBusAddress (cb, pBusAddressRet): ercType

where

cb

specifies the maximum length that may be supplied in subsequent RemapBusAddress operations on the bus address returned.

pBusAddressRet

is the address of a double word to which the calculated bus address is returned.

Request Block

ReserveBusAddress is a system-common procedure.

ReservePartitionMemory (cb, fLongLived, pMhRet): ercType

Description

ReservePartitionMemory dynamically allocates additional memory in an application partition and returns a memory handle that uniquely identifies this memory.

ReservePartitionMemory may only be called by a program executing in the single application partition (primary partition) in memory. The additional memory actually is not allocated until the caller exits and the application partition is recreated. The memory initially is set to zeros and optionally may be set to zeros every time the partition terminates by specifying FALSE for the *fLongLived* parameter.

ReservePartitionMemory typically is used by a system service that requires additional memory for each user number to be served. Note that the amount of memory specified is only the amount needed for each user number, not the maximum memory required for all user numbers. During system service installation, the system service must call ReservePartitionMemory before calling ConvertToSys.

To access the memory reserved in the partition, the caller passes the memory handle returned as the value of the *structCode* parameter to the GetPStructure operation along with the partition user number. (See GetPStructure.)

The memory allocated belongs to the partition (user number) and temporarily becomes unavailable when the partition terminates or is swapped out to disk. The memory always is available for system services that access it synchronously with servicing a request: GetPStructure should be called after the client requests the service. Because the memory address returned by GetPStructure becomes invalid when the application terminates, system services should treat such memory addresses as short–lived.

Procedural Interface

ReservePartitionMemory (cb, fLongLived, pMhRet): ercType

where

cb

is the count of bytes to be reserved.

fLongLived

FALSE means the memory is reset to zeros when the partition containing the memory terminates.

pMhRet

is the address at which the memory handle is returned.

Request Block

sMhRet is always 2.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	354
12	cb	2	
14	fLongLived	2	
16	reserved	2	
18	pMhRet	4	
22	sMhRet	2	2

This page intentionally left blank

ResetCommLine (commLineHandle): ercType

Description

ResetCommLine makes the specified communications channel on a serial communications controller, such as the 8274, available for use again after the channel has been initialized by InitCommLine. No further interrupts will be routed to the caller who did the InitCommLine. If an interrupt is received after ResetCommLine is executed, but before another InitCommLine, the channel is reset.

See "Serial Ports" in the *CTOS Operating System Concepts Manual.*)

Procedural Interface

ResetCommLine (commLineHandle): ercType

where

commLineHandle

is a handle returned by InitCommLine.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	4103
12	unused	6	
18	commLineHandle	2	

ResetDeviceHandler (tyDev): ercType

Description

ResetDeviceHandler sets up the default device handler for a specified device type. This operation permits an application to remove a device handler that was established by a call to SetDeviceHandler. If *tyDev* is FFFFh, ResetDeviceHandler removes all interrupt handlers and all device handlers established in previous calls to SetDeviceHandler and SetIntHandler.

For details on interrupt handlers, see "Interrupt Handlers" in the *CTOS Operating System Concepts Manual.*

Procedural Interface

ResetDeviceHandler (tyDev): ercType

where

tyDev

is the type of device (word) to which the default handler is assigned. If *tyDev* is FFFFh, all interrupt handlers and device handlers established by the application will be removed.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	469
12	tyDev	2	

ResetFrame (iFrame): ercType

Description

ResetFrame restores the specified frame to its initial state, that is, all character positions are blanked and all character attributes are reset. The visible cursor of the frame is disabled (the *iLineCursor* and *iColCursor* fields of the frame descriptor of the Video Control Block are set to 255). In addition, the frame descriptor fields *iLineLeftOff* and *iColLeftOff* are set to 255. (For a description of the frame descriptor and the Video Control Block, see "Video" in the *CTOS Operating System Concepts Manual*. The formats of these structures are shown in Chapter 4, "System Structures.")

If pausing is enabled, ResetFrame resets the frame descriptor field, *iLinePause*, to scroll the maximum number of frame lines before the NEXT PAGE or SCROLL UP message is displayed.

Procedural Interface

ResetFrame (iFrame): ercType

where

iFrame

specifies the frame. A value of 255 is treated as a special case and is reserved for internal use only.

Request Block

ResetFrame is a system-common procedure.

This page intentionally left blank.

ResetIBusHandler (wId, pProc): ercType

Description

ResetIBusHandler allows the caller to disconnect an I–Bus handler from a specific I-Bus device that the handler was registered with using SetIBusHandler. (See SetIBusHandler for a detailed description of the arguments to the I-Bus device handler.)

Procedural Interface

ResetIBusHandler (wId, pProc): ercType

where

wId

is the 2 byte I–Bus device identifier used in SetIBusHandler.

pProc

is the Global Descriptor Table (GDT) based memory address of the procedure. Typically, this procedure is part of the service that handles the specific I–Bus device. The procedure at *pProc* is called from an interrupt level with a byte of data from the I–Bus device.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	2
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	349
12	wId	2	
14	pProc	4	

ResetMemoryLL: ercType

Description

ResetMemoryLL deallocates all long-lived memory within a partition.

Note that executing this operation will cause the Executive, which depends on part of the contents of long-lived memory, to loose the ability to redo the previous command.

Procedural Interface

ResetMemoryLL: ercType

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	0
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	47

This page intentionally left blank

ResetPSCounters (sh, pbBlockID, cbBlockID): ercType

Description

ResetPSCounters is a synonym for PSResetCounters. See the description of PSResetCounters for details.

This page intentionally left blank

ResetTimerInt (pTpib): ercType

Description

The ResetTimerInt primitive terminates the Timer Pseudointerrupt Block (TPIB) initiated by a previous SetTimerInt call. ResetTimerInt is used only to cancel a previous SetTimerInt operation *before* the requested pseudointerrupt has occurred. The "No such TPIB" status code is returned if the pseudointerrupt has already occurred.

(See Chapter 4, "System Structures," for the format of a TPIB.)

Procedural Interface

ResetTimerInt (pTpib): ercType

where

pTpib

is the memory address of the TPIB to be terminated.

Request Block

ResetTimerInt is a Kernel primitive.

This page intentionally left blank

ResetTrapHandler (iInt): ercType

Description

ResetTrapHandler sets up the default trap handler for a specified interrupt level. This operation permits an application to remove a trap handler that was established by a call to SetTrapHandler or Set386TrapHandler.

For details on interrupt handlers, see "Interrupt Handlers" in the *CTOS Operating System Concepts Manual.*

Procedural Interface

ResetTrapHandler (iInt): ercType

where

iInt

is the interrupt level (0 to 255).

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	461
12	iInt	2	

ResetVideo (nCols, nLines, fAttr, bSpace, psMapRet): ercType

Description

ResetVideo suspends video refresh and changes the values stored in the Video Control Block (VCB) to reflect the specified parameters. All frames and all VCB fields (except *fReverseVideo*) are initialized. Additionally, the *fExecScreen* flag in the Application System Control Block is set to FALSE. Subsequent calls to the InitVidFrame operation are validated against the values in the VCB. (See Chapter 4, "System Structures," for the format of the VCB.)

When writing software that must work on several workstation models, use QueryVidHdw or QueryVideo to determine the level of video capability before calling the ResetVideo operation.

On character-mapped workstations with Enhanced Video (EV), ResetVideo may cause a new font to be loaded. For EV, the display modes (number of columns by number of lines) and their fonts are as described below:

Display Mode	**Cell Size**	**Character Size**	**Default System Font**
80x29	9x12	7x10	T1Sys.font
80x34	9x10	7x8	80x34_CharSys.font
132x29	7x12	5x9	132x29_CharSys.font
132x34	7x10	5x8	132x34_CharSys.font

Applications that use their own fonts with EV also require a different font if they plan to use any display mode other than the default 80x29 mode. The Video System Service retains the name of the font file specified in a call to LoadFontRam. When the display mode changes, however, ResetVideo causes a different font file to be loaded. Depending on the display mode, a prefix is concatenated with the name of the client's original font file name to create the new file name. The prefixes are as follows:

Display Mode	**Prefix**
80x29	
80x34	80x34_
132x29	132x29_
132x34	132x34

If, for example, the client's font file name is [Sys]<Sys>Special.font, and the client calls ResetVideo to change the mode to 132x29, the following font file is loaded:

[Sys]<Sys>132x29_Special.font

If the file is not found, status code 203 ("File not found") is returned.

NOTE: This font loading will only occur with system fonts. If users elect to use their own special fonts, the Video Access Method does not attempt to load fonts on mode switches. When using User Defined fonts, it is the responsibility of the user to ensure that the appropriate font file is loaded (see LoadFontRam).

Procedural Interface

ResetVideo (nCols, nLines, fAttr, bSpace, psMapRet): ercType

where

nCols

specifies the number of characters per line. (See QueryVidHdw.)

nLines

specifies the number of lines per screen. (See QueryVidHdw.) A value of 255 is treated as a special case and is reserved for internal use only.

fAttr

specifies whether the character map is to include character attributes. It is TRUE if character attributes are to be used; otherwise it is FALSE.

bSpace

specifies a character code that is blank in the font. This is used when the character map is initialized by the InitCharMap, ResetFrame, and ScrollFrame operations. (For more information, see "Video" in the *CTOS Operating System Concepts Manual.*)

psMapRet

is the memory address of the word to which the required size of the character map is returned.

Request Block

ssMapRet is always 2.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	74
12	nCols	1	
13	nLines	1	
14	fAttr	1	
15	bSpace	1	
16	reserved	2	
18	psMapRet	4	
22	ssMapRet	2	2

ResetVideoGraphics (nCols, nLines, fAttr, bSpace, psMapRet, nPixelsWide, nPixelsHigh, nPlanes, fBackgroundColor): ercType

Caution: *This operation works on workstation hardware only. Do not use it on shared resource processor hardware.*

NOTES:

1. *This operation is supported by protected mode operating systems only.*
2. *When using ResetVideoGraphics to change the column format of the character map or the resolution of the bit map, your program also must call PdSetVirtualCoordinates or PdSetCharMapVirtualCoordinates to reset the virtual coordinates of the pointing device. For details on these operations, see the* CTOS Programming Guide.

Description

ResetVideoGraphics functions in the same way as the ResetVideo operation but allows the caller more options. (See the description of ResetVideo.) With ResetVideoGraphics the caller can, in addition, set the graphics bit map resolution, the number of planes for color, and the background color mode on workstations with VGA capability.

The values you specify for *nCols*, *nLines*, *nPixelsWide*, and *nPixelsHigh* are used by this operation to select a display mode that can minimally support them and will be at least the sizes specified. If this cannot be done, however, status code 501 ("Invalid argument to VDM") is returned. Any value can be 0, in which case the value will not be used to decide on a mode. If all values are 0, a default will be chosen that is the same as the initial mode of the monitor at boot time.

Following are examples:

nCols	80
nLines	29
nPixelsWide	0
nPixelsHigh	0
nPlanes	0

With the above values, ResetVideoGraphics selects a monitor resolution of 80 columns by 29 lines and an appropriate width and height in pixels for the graphics resolution. In addition, the operation selects an appropriate number of planes (because 0 is specified for *nPlanes*).

nCols	0
nLines	0
nPixelsWide	720
nPixelsHigh	348
nPlanes	4

With the above values, ResetVideoGraphics selects an appropriate monitor resolution, 720 pixels wide by 348 pixels high for the graphics resolution, and 4 planes. If you specified a graphics resolution greater than that supported by the monitor (such as 1024 by 768 pixels for a monitor that only supports 720 x 348), status code 501 is returned. If, however, you specified a graphics resolution less than the highest resolution supported by the monitor (such as 720 x 348 pixels for a monitor that supports 1024 x 768), ResetVideoGraphics will accept the lower pixel values.

nCols	80
nLines	29
nPixelsWide	1024
nPixelsHigh	348
nPlanes	4

With the above values, ResetVideoGraphics selects a mode that is *at least* 80 columns by 29 lines and 1024 pixels wide by 348 pixels high. Say, for example, the monitor supported the following two modes:

80 x 29 and 720 x 348

and

80 x 38 and 1024 x 768

In this case, ResetVideoGraphics would select the latter mode.

Procedural Interface

ResetVideoGraphics (nCols, nLines, fAttr, bSpace, psMapRet, nPixelsWide, nPixelsHigh, nPlanes, fBackgroundColor): ercType

where

nCols

is a byte that specifies the number of characters per line. (See QueryVidHdw.) If the value of *nCols* is 0, this parameter is ignored, and any character width that can accomodate the resolution can be chosen.

nLines

is a byte that specifies the number of lines per screen. (See QueryVidHdw.) If the value of *nLines* is 0, this parameter is ignored, and any character height that can accomodate the resolution can be chosen. A value of 255 is treated as a special case and is reserved for internal use only.

fAttr

is a flag (byte) that is TRUE if the character map is to include character attributes.

bSpace

is a byte that specifies a character code that is blank in the font. This is used when the character map is initialized by the InitCharMap, ResetFrame, or ScrollFrame operation. (For more information, see "Video" in the *CTOS Operating System Concepts Manual*.)

psMapRet

is the memory address of the word to which the required size of the character map is returned.

nPixelsWide

is the number of pixels wide (word) for the bit map. If the value of *nPixelsWide* is 0, this parameter is ignored, and any horizontal resolution that can accomodate the character map can be chosen.

nPixelsHigh

is the number of pixels high (word) for the bit map. If the value of *nPixelsHIgh* is 0, this parameter is ignored, and any vertical resolution that can accomodate the character map can be chosen.

nPlanes

is the number of planes (byte) used for color graphics. If the value of *nPlanes* is 0, this parameter is ignored, and an appropriate number of planes is chosen for this resolution.

fBackgroundColor

is a flag (byte) that sets the background color mode. If *fBackgroundColor* is FALSE, all character backgrounds are color index 0. If *fBackgroundColor* is TRUE, character backgrounds are indexes 0 through 7, matching the value of the color bits in the attribute byte. (For details on color programming, see "Using Color" in the *CTOS Programming Guide*.)

Request Block

ssMapRet is always 2.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	10
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	33137
12	nCols	1	
13	nLines	1	
14	fAttr	1	
15	bSpace	1	
16	nPixelsWide	2	
18	nPixelsHIgh	2	
20	nPlanes	1	
21	fBackgroundColor	1	
22	psMapRet	4	
26	ssMapRet	2	2

This page intentionally left blank

ResetXBusMIsr (ih): ercType

Caution: *This operation works on workstation hardware only. Do not use it on shared resource processor hardware.*

Description

ResetXBusMIsr purges an interrupt handler previously established using SetXBusMIsr.

Procedural Interface

ResetXBusMIsr (ih): ercType

where

ih

is the interrupt handle returned by SetXBusMIsr.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	2
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	32800*
12	ih	2	

*On BTOS II, 3.2 workstation operating systems, the request code 8205 is also supported.

ResizeIoMap (maxPortAddr, pIoMapRet): ercType

Description

ResizeIoMap changes the size of the 386 I/O permission bit map that is located in the Task State Segment (TSS) of the calling program. In addition, ResizeIoMap returns the linear memory address of the resized bit map.

An I/O permission bit map specifies whether the application can access each I/O address. In this map, a bit value of 0 indicates that access permission is granted; a bit value of 1 indicates that access pemission is denied. (For details on the I/O permission bit map, see the *80386 Programmer's Reference Manual.*)

If ResizeIoMap moves the permission bit map, it copies the contents of the current bit map to the new location. When the map size is increased, each bit in the added portion is set to 1 (permission denied).

Procedural Interface

ResizeIoMap (maxPortAddr, pIoMapRet): ercType

where

maxPortAddr

is the highest I/O address to be accessed by the calling application. This value can range from 0 (no permission bit map) to FFFFh (the maximum allowable size of the permission bit map).

pIoMapRet

is the memory address of a double word where the linear address of the resized bit map is returned.

Request Block

ResizeIoMap is a system-common procedure.

ResizeSegment (sn, qBytes, pRa32Ret): ercType

Note: This operation only works on virtual memory operating systems and is for use by programs executing in protected mode.

Description

ResizeSegment changes the size of the specified segment. The segment to be resized must have been previously created by the AllocateSegment operation. (See AllocateSegment.)

Procedural Interface

ResizeSegment (sn, qBytes, pRa32Ret): ercType

where

sn

is the selector (word) of the segment to resize.

qBytes

is the new segment size (dword) rounded up to the next 4K byte boundary.

pRa32Ret

is the memory address of the location to which the 32-bit offset of the resized segment is returned.

Request Block

sRa32Ret is always 4.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	0
3	nRespPbCb	1	1
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	434
12	sn	2	
14	qBytes	4	
18	pRa32Ret	4	
22	sRa32Ret	2	4

Respond (pRq): ercType

Description

The Respond primitive is used only by a system service to respond to a client. After the system service has completed the processing of a service request, it invokes Respond to send the request block of the client back to the response exchange specified in the request block.

The Respond primitive accepts the memory address of the request block of the client as its only parameter; the system service must use the same memory address as a parameter to the Respond primitive that the client used as a parameter to the Request primitive. The exchange to which the response is directed is determined by the exchange response field of the request block.

Calls to the Respond primitive must exactly match calls to the Request primitive; that is, each respond must answer a request and each request must be answered by a respond.

Procedural Interface

Respond (pRq): ercType

where

pRq

is the memory address of the same request block that the system service process received from its exchange.

Request Block

Respond is a Kernel primitive.

This page intentionally left blank

ResumeContext: ercType

Description

ResumeContext is used by an asynchronous system service to resume the execution of a context at the point where it left off the last time the system service made an asynchronous request to an external agent. If a request was built on the stack, the request block is removed, and the status code is stored in a global variable.

Procedural Interface

ResumeContext: ercType

Request Block

ResumeContext is an object module procedure in Async.lib.

This page intentionally left blank

ReuseAlias (snAlias, pSource): ercType

Caution: *This operation is supported in protected mode only.*

Description

ReuseAlias updates the segment descriptor corresponding to *snAlias*. After the update, *snAlias*, combined with an offset of 0, references the same linear address as *pSource*.

The CreateAlias operation initially allocates *snAlias*.

ReuseAlias supports software that requires the sequential use of a large number of segment aliases.

Procedural Interface

ReuseAlias (snAlias, pSource): ercType

where

snAlias

is the alias selector of the specified source memory address, *pSource*.

pSource

is the memory address to which the alias is created.

Request Block

ReuseAlias is a system-common procedure.

This page intentionally left blank

ReuseAliasLarge (snAlias, snSource, dOffsetSource, wLimit): ercType

Caution: *ReuseAliasLarge is supported only by protected mode operating systems. Before using this operation, call CurrentOsVersion to check the operating system version.*

Description

ReuseAliasLarge replaces the base address in the descriptor specified by *snAlias* with an address computed by adding an offset (*dOffsetSource*) to the base address in the descriptor specified by *snSource*. ReuseAliasLarge also replaces the limit field of *snAlias* with the limit specified by *wLimit*.

ReuseAliasLarge succeeds only if the new base and limit fall within the same hypersegment, which must be associated with the calling program. (A hypersegment is a contiguous memory region containing information such as long-lived memory, short-lived memory, or code.)

ReuseAliasLarge differs from ReuseAlias in that ReuseAliasLarge allows an offset and a limit to be specified. (See ReuseAlias.)

ReuseAliasLarge, together with CreateAlias, can be used to build alias pointers to address memory within a large data object (greater than 64k bytes) that was alloscated using AllocAllMemorySL. (An example showing how to do this is shown in "Writing Compatible Programs" in the *CTOS Programming Guide*.)

Procedural Interface

ReuseAliasLarge (snAlias, snSource, dOffsetSource, wLimit): ercType

where

snAlias

is the alias selector to reuse.

snSource

is the selector specifying the base address of the new alias.

dOffsetSource

is the 32-bit offset from the base address of the new alias.

wLimit

is the limit of the new alias.

Request Block

ReuseLargeAlias is a system–common procedure.

RewriteMarkedQueueEntry (pbQueueName, cbQueueName, qeh, pEntry, sEntry): ercType

Description

RewriteMarkedQueueEntry is used by the queue server to rewrite the specified queue entry with a new entry in the specified queue. RewriteMarkedQueueEntry can be called, for example, to update a field contained in a queue entry. The entry to be overwritten is identified by a queue entry handle returned from a previous MarkKeyedQueueEntry or MarkNextQueueEntry operation.

Procedural Interface

RewriteMarkedQueueEntry (pbQueueName, cbQueueName, qeh, pEntry, sEntry): ercType

where

pbQueueName
cbQueueName

describe a queue name corresponding to a queue name specified when the queue was installed.

qeh

is the 32-bit queue entry handle returned from a previous call to the MarkKeyedQueueEntry or MarkNextQueueEntry operation.

pEntry
sEntry

describe the buffer where the type-specific portion of the queue entry is read.

Request Block

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	4
1	RtCode	1	0
2	nReqPbCb	1	2
3	nRespPbCb	1	0
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	145
12	qeh	4	
16	pbQueueName	4	
20	cbQueueName	2	
22	pEntry	4	
26	sEntry	2	

RgParam (iParam, jParam, pSdRet): ercType

Description

RgParam provides access to the parameters stored in the Variable Length Parameter Block (VLPB).

The VLPB can be described as a two-dimensional array of strings where each array element is a value of a subparameter (*jParam*). Each row is a parameter (*iParam*). (See Chapter 4, "System Structures," for the format of the Variable Length Parameter Block.)

The Executive is an example of a program that uses RgParam. Each line of an Executive command form corresponds to an *iParam*. Each of several user entries on an Executive command line, such as a file name entered on the File list line of the Files command, corresponds to a *jParam*.

RgParam returns the memory address and size of a subparameter (*jParam*).

For example, in the Executive **Files** command form below, the single word "yes" is accessed by RgParam (2,0). The file name, SysImage.sys, is accessed by RgParam (1,1). Subparameters (*jParams*) on the same command line (*iParam*) are delimited by one or more spaces. The Executive stores the command name used to invoke the application program (Files, in this example) in RgParam (0,0).

Command Form

```
Files
    [File list]              CrashDump.sys  SysImage.sys
    [Details?]               yes
    [Print file]
    [Suppress sort]
    [Max columns]
```

Procedural Interface

RgParam (iParam, jParam, pSdRet): ercType

where

iParam

is the index of the parameter.

jParam

is the index of the subparameter.

pSdRet

is the location of a 6-byte block of memory. The memory address of *jParam* is returned in the first 4 bytes, and its size is stored in the last 2 bytes.

Request Block

RgParam is an object module procedure.

RgParamInit (pVarParams, sVarParams, iParamMax): ercType

Description

RgParamInit initializes the specified memory to be the Variable Length Parameter Block (VLPB). If the block of memory is not large enough to contain the VLPB, RgParamInit increases its size by allocating additional long-lived memory.

Caution: *If RgParamInit must allocate additional long-lived memory for the VLPB, existing long-lived memory allocated at higher addresses is overwritten. For this reason, applications should be sure adequate memory is allocated.*

Procedural Interface

RgParamInit (pVarParams, sVarParams, iParamMax): ercType

where

pVarParams
sVarParams

describe the block of memory to be used for the Variable Length Parameter Block. If *sVarParams* is 0, *pVarParams* is ignored and the current Variable Length Parameter Block is reinitialized.

iParamMax

is one less than the number of parameters to be recorded.

Request Block

RgParamInit is an object module procedure.

RgParamSetEltNext (pSd): ercType

Description

RgParamSetEltNext creates an additional subparameter of the current parameter in the Variable Length Parameter Block. A call to RgParamSetEltNext must follow a call to either RgParamSetListStart or RgParamSetEltNext with no other RgParam types of calls being made in between. This is because RgParamSetListStart sets the global variable for placing subparameters within the Variable Length Parameter Block. RgParamSetEltNext uses this global variable.

If the Variable Length Parameter Block is not large enough to accommodate this subparameter, it is compacted and an attempt is made to extend it by allocating additional long-lived memory. This attempt succeeds only if the Variable Length Parameter Block is at the top of the long-lived memory of an application partition.

Procedural Interface

RgParamSetEltNext (pSd): ercType

where

pSd

is the location of a 6-byte block of memory, the first 4 bytes of which contain the memory address of the string to be used and the last bytes of which contain the string's length.

Request Block

RgParamSetEltNext is an object module procedure.

This page intentionally left blank

RgParamSetListStart (iParam): ercType

Description

RgParamSetListStart initiates the creation of a parameter with multiple subparameters. RgParamSetEltNext, which must be called immediately following the invocation of RgParamSetListStart, creates a subparameter. If the parameter already exists, all its old subparameters are destroyed and the memory they occupied is reused.

Procedural Interface

RgParamSetListStart (iParam): ercType

where

iParam

is the index of the parameter.

Request Block

RgParamSetListStart is an object module procedure.

This page intentionally left blank

RgParamSetSimple (iParam, pSd): ercType

Description

RgParamSetSimple creates a parameter with one subparameter. If the parameter already exists, all its old subparameters are destroyed and the memory they occupied reused.

If the Variable Length Parameter Block is not large enough to accommodate this parameter, it is compacted and an attempt made to extend it by allocating additional long-lived memory. This attempt succeeds only if the Variable Length Parameter Block is at the top of the long-lived memory of an application partition.

Procedural Interface

RgParamSetSimple (iParam, pSd): ercType

where

iParam

is the index of the parameter.

pSd

is the location of a 6-byte block of memory, the first 4 bytes of which contain the memory address of the string to be used and the last 2 bytes of which contain the string's length.

Request Block

RgParamSetSimple is an object module procedure.

This page intentionally left blank

RkvsVersion (pbSpec, cbSpec, pVerStruct, sVerStruct, pcbRet): ercType

Description

RkvsVersion returns a structure that provides the version of the Remote Keyboard/Video Service (RKVS) and indicates whether the Remote User Manager (RUM) is being used. (RKVS is also known as **Cluster View**. For details on the **Cluster View** utility, see the *CTOS Executive Reference Manual*.)

Procedural Interface

RkvsVersion (pbSpec, cbSpec, pVerStruct, sVerStruct, pcbRet): ercType

where

pbSpec
cbSpec

describe the location (node name and device name of the processor board) where the RKVS service is executing, for example [FP01] or {Foo}[GP00]

pVerStruct
sVerStruct

describe the memory area to which the structure containing the version and RUM flag value is written. The format of the structure is as follows:

Offset	**Field**	**Size (Bytes)**
0	version	2
2	fRumPresent	1

pcbRet

describe a word into which the length of the returned structure is placed.

Request Block

scbRet is always 2.

Offset	Field	Size (Bytes)	Contents
0	sCntInfo	1	6
1	RtCode	1	0
2	nReqPbCb	1	1
3	nRespPbCb	1	2
4	userNum	2	
6	exchResp	2	
8	ercRet	2	
10	rqCode	2	8208
12	reserved	6	
18	pbSpec	4	
22	cbSpec	2	
24	pVerStruct	4	
28	sVerStruct	2	
30	pcbRet	4	
34	scbRet	2	2

RsrcCopyFromFile (pRwaTarget, pbSourceDesc, cbSourceDesc, fOverwriteOk): ercType

Description

RsrcCopyFromFile copies the resource descriptor from a source file to the specified target run file resource work area (RWA). The target run file RWA should already be established with the RsrcInitSetAccess operation before RsrcCopyFromFile is called. (See RsrcInitSetAccess.) The resource data is not actually copied to the target run file until RsrcSetEndAccess is called on the target file resource set or RsrcEndSession is called to end the resource session. (See RsrcSetEndAccess and RsrcEndSession.)

The resource information in the descriptor passed to RsrcCopyFromFile (*pbSource/cbSource*) can be from either of two sources: a CTOS data file (for example, a symbol file) or a memory buffer of descriptors obtained by another resource access operation. Each of these sources is discussed below.

RsrcCopyFromFile is the only operation that can be used to copy resources from CTOS data files. Data files do not have an RWA established for them by the RsrcInitSetAccess operation. Therefore, the resource information described by *pbSource/cbSource* must be formatted by the caller. To enable RsrcCopyFromFile to access the data file and the resource in it, at minimum the following information must be provided:

File handle of the data file containing the resource
Flag data specifying the file handle type
Logical file address of the resource
Size in bytes of the resource
Resource type and ID

(For details on the format of a resource descriptor, see "Resource Descriptor" in the *CTOS Procedural Interface Reference Manual* section entitled "System Structures.")

The resource information described by *pbSource/cbSource* also can be a descriptor copied to a memory buffer by the RsrcGetSetTypeInfo or the RsrcGetDesc operation. (See the descriptions of RsrcGetSetTypeInfo and RsrcGetDesc for details.)

If a resource descriptor with the same resource ID and type is already present in the target run file RWA and *fOverwriteOk* is TRUE, RsrcCopyFromFile overwrites that resource descriptor with the new one. If *fOverwriteOk* is FALSE under the same circumstances, status code 52108 ("Resource already exists") is returned. Status code 52112 ("Cannot modify resource set") is returned if the target file was not opened in modify mode.

For details on the operations for managing resources in disk files, see "Accessing Resources in Disk Files" in the *CTOS Operating System Concepts Manual* section entitled "Utility Operations."

Procedural Interface

RsrcCopyFromFile (pRwaTarget, pbSourceDesc, cbSourceDesc, fOverwriteOk): ercType

where

pRwaTarget

is the memory address of the of the target file RWA.

pbSourceDesc
cbSourceDesc

describe the area holding the resource information from the (source) data file. The information must be in the format of a resource descriptor.

fOverWriteOk

is a flag (word). TRUE allows RsrcCopyFromFile to overwrite an existing version of the specified resource in the target file RWA with the one provided in this operation.

Request Block

RsrcCopyFromFile is an object module procedure.

This page intentionally left blank.

RsrcCopyFromRsrcSet (pRwaTarget, pRwaSource, typeCode, rsrcId, fOverwriteOk): ercType

Description

RsrcCopyFromRsrcSet copies the resource descriptor with the given type and ID from the specified source file resource work area (RWA) to the specified target file RWA. The resource itself is not actually copied anywhere until either the RsrcEndSetAccess or the RsrcSessionEnd operation is called. (See RsrcEndSetAccess and RsrcSessionEnd.)

When a resource descriptor with the same resource ID and type is already present in the target file RWA, and *fOverwriteOk* is TRUE, RsrcCopyFromRsrcSet overwrites the existing resource descriptor with the new one. Under the same circumstances, where *fOverwriteOk* is FALSE, status code 52108 ("Resource already exists") is returned. In cases where the target file is not opened in modify mode, status code 52112 ("Cannot modify resource set") is returned. If the source RWA does not have a resource with the specified type and ID, status code 52106 ("No such resource") is returned.

For details on the operations for managing resources in disk files, see "Accessing Resources in Disk Files" in the *CTOS Operating System Concepts Manual* section entitled "Utility Operations."

Procedural Interface

RsrcCopyFromRsrcSet (pRwaTarget, pRwaSource, typeCode, rsrcId, fOverwriteOk): ercType

where

pRwaTarget

is the memory address of the target file RWA.

pRwaSource

is the memory address of the source file RWA.

typeCode

is the type code (word) of the resource to be copied.

rsrcId

is the resource ID (word).

fOverWriteOk

is a word. TRUE allows RsrcCopyFromRsrcSet to overwrite the existing version of the resource with the new one.

Request Block

RsrcCopyFromRsrcSet is an object module procedure.

RsrcCopyRestRunfile (pRwaTarget, pRwaSource): ercType

Description

RsrcCopyRestRunfile copies information on the location and size of the source run file contents (resources excepted) to the target run file resource work area (RWA). The run file contents are not actually copied to the target run file until RsrcSetEndAccess or RsrcEndSession is called. (See RsrcSetEndAccess and RsrcEndSession.)

For details on the operations for managing resources in disk files, see "Accessing Resources in Disk Files" in the *CTOS Operating System Concepts Manual* section entitled "Utility Operations."

Procedural Interface

RsrcCopyRestRunfile (pRwaTarget, pRwaSource): ercType

where

pRwaTarget

is the memory address of the target file resource work area (RWA).

pRwaSource

is the memory address of the source file RWA.

Request Block

RsrcCopyRestRunfile is an object module procedure.

This page intentionally left blank.

RsrcDelete (pRwa, typeCode, rsrcId): ercType

Description

RsrcDelete deletes the resource descriptor with the given *typeCode* and *ID* from the resource set specified by *pRwa*. The resource itself is not actually deleted from the resource set in the disk file until either the RsrcEndSetAccess or the RsrcSessionEnd operation is called.

In the case where the file whose resource work area (RWA) is being modified was not previously opened in modify mode, status code 52112 ("Cannot Modify Resource Set") is returned. If the file does not contain the specified resource, status code 52106 ("No such resource") is returned.

For details on the operations for managing resources in disk files, see "Accessing Resources in Disk Files" in the *CTOS Operating System Concepts Manual* section entitled "Utility Operations."

Procedural Interface

RsrcDelete (pRwa, typeCode, rsrcId): ercType

where

pRwa

is the memory address of the RWA.

typeCode

is the type code (word) of the resource to be deleted.

rsrcId

is the resource ID (word).

Request Block

RsrcDelete is an object module procedure.

RsrcEndSetAccess (pRwa, fileType, fSave): ercType

Description

RsrcEndSetAccess checks to see if any resource descriptors from the specified RWA have been copied to other RWAs. If so, the resources associated with this RWA whose descriptors have been copied are themselves copied to a temporary file, and the logical file addresses of these resources in the descriptors in the other RWAs are adjusted to reflect the new temporary file addresses. If any resource descriptors from other RWAs have been copied to this RWA, all the resources in this RWA are copied to the resource set in the associated file on disk. If, in addition, RsrcCopyRestRunFile has been called on this RWA, the nonresource parts of the run file also are copied to the new run file.

RsrcEndSetAccess should be called periodically by applications that allow the end user to keep files open for any length of time. Otherwise the contents of the in-memory RWA can be lost if a machine malfunction occurs. RsrcEndSetAccess should be called again when an application is finished updating the resource set for a file. If it is not, however, and the specified RWA still has outstanding entries, the call to the RsrcSessionEnd operation writes out any outstanding resources to the appropriate disk files. (See RsrcSessionEnd.)

If no file type is specified, or if the file type specified is erroneous, status code 52117 ("No file type specified") is returned.

For details on the operations for managing resources in disk files, see "Accessing Resources in Disk Files" in the *CTOS Operating System Concepts Manual* section entitled "Utility Operations."

Procedural Interface

RsrcEndSetAccess (pRwa, fileType, fSave): ercType

where

pRwa

is the memory address of the RWA to be deallocated.

fileType

is a value (word) indicating the disk file type to be created if the file whose RWA is being deallocated has been modified. The values and meanings of *fileType* are

Value	**Meaning**
2	Binary resource file
4	Run file

fSave

is a flag (word). If *fSave* is TRUE, RsrcEndSetAccess updates the resources in the resource set of the associated disk file according to the changes in the RWA.

Request Block

RsrcEndSetAccess is an object module procedure.

RsrcGetAllSetTypeInfo (pRwa, cbPerDesc, pbDescBuf, cbDescBuf, pcDescRet): ercType

Description

RsrcGetAllSetTypeInfo writes the descriptor of every resource in the resource set identified by *pRwa* in the buffer provided into the same order that they are in the RWA. The caller must specify the memory address and size of the buffer to contain the descriptors and the number of bytes in each descriptor to be written. To calculate the buffer size, the caller can use the RsrcGetCountAllSetTypes operation. (See RsrcGetCountAllSetTypes.)

RsrcGetAllSetTypeInfo differs from RsrcGetSetTypeInfo in that the latter operation only returns resource descriptors of a given type.

In some cases, *cbDesBuf* may not be large enough to hold all resources. If so, status code 52116 ("Buffer overflow") is returned. If no resources are specified, status code 52106 ("No such resource") is returned.

For details on the operations for managing resources in disk files, see "Accessing Resources in Disk Files" in the *CTOS Operating System Concepts Manual* section entitled "Utility Operations."

Procedural Interface

RsrcGetAllSetTypeInfo (pRwa, cbPerDesc, pbDescBuf, cbDescBuf, pcDescRet): ercType

where

pRwa

is the memory address of the resource work area (RWA).

cbPerDesc

is the number of bytes to be written for each descriptor.

pbDescBuf
cbDescBuf

desribe the buffer to which the resource descriptor(s) are written. For details on the resource descriptor format, see "Resource Descriptor" in the *CTOS Procedural Interface Reference Manual* section entitled "System Structures."

pcDescRet

is the memory address of a word to which the count of resources is written.

Request Block

RsrcGetAllSetTypeInfo is an object module procedure.

RsrcGetCountAllSetTypes (pRwa, pcRsrc): ercType

Description

RsrcGetCountAllSetTypes returns the number of resources in the resource set specified by *pRwa*. An application can use the count returned to calculate the size of the buffer it needs to pass to RsrcGetAllSetTypeInfo to obtain all the resource descriptors in the RWA. (See RsrcGetAllSetTypeInfo.)

RsrcGetCountAllSetTypes differs from RsrcGetCountSetType in that the latter operation returns the number of resources of a given type that reside in a file.

If the file associated with the resource work area (RWA) does not contain any resources, status code 52105 ("No resources") is returned.

For details on the operations for managing resources in disk files, see "Accessing Resources in Disk Files" in the *CTOS Operating System Concepts Manual* section entitled "Utility Operations."

Procedural Interface

RsrcGetCountAllSetTypes (pRwa, pcRsrc): ercType

where

pRwa

is the memory address of the resource work area (RWA).

pcRsrc

is the memory address to which the count of resources is returned.

Request Block

RsrcGetCountAllSetTypes is an object module procedure.

RsrcGetCountSetType (pRWA, typeCode, pcRsrc): ercType

Description

RsrcGetCountSetType returns the number of resources of the specified type in the resourse set identified by *pRwa*.

RsrcGetCountSetType differs from RsrcGetCountAllSetType in that the latter operation returns the total number of resources within a file.

If the file associated with the resource work area (RWA) does not contain any resources, status code 52105 ("No resources") is returned. If it does not contain resources of the specified type, status code 52106 ("No such resource") is returned.

For details on the operations for managing resources in disk files, see "Accessing Resources in Disk Files" in the *CTOS Operating System Concepts Manual* section entitled "Utility Operations."

Procedural Interface

RsrcGetCountSetType (pRwa, typeCode, pcRsrc): ercType

where

pRwa

is the memory address of the RWA.

typeCode

is the type code (word) of the resource.

pcRsrc

is the memory address of a word to which the count of resources of the given type is written.

Request Block

RsrcGetCountSetType is an object module procedure.

RsrcGetDesc (pRwa, typeCode, rsrcId, pbDesc, cbDescMax, pcbDescRet): ercType

Description

RsrcGetDesc returns the following information in the resource descriptor format provided by the caller:

File handle of the file containing the resource
Flag data specifying the file handle type
Logical file address of the resource
Size in bytes of the resource
Resource type and ID

For details on the format of the resource descriptor, see "Resource Descriptor" in the *CTOS Procedural Interface Reference Manual* section entitled "System Structures."

If the specified resource is not present in the file associated with the resource work area (RWA), status code 52106 ("No such resource") is returned. In such a case, the data written in the lfa field of the resource descriptor is the location where this resource would be added if it actually was in the file.

RsrcGetDesc differs from RsrcGetSetTypeInfo in that the latter operation returns all the resource descriptors of a given type.

For details on the operations for managing resources in disk files, see "Accessing Resources in Disk Files" in the *CTOS Operating System Concepts Manual* section entitled "Utility Operations."

Procedural Interface

RsrcGetDesc (pRwa, typeCode, rsrcId, pbDesc, cbDescMax, pcbDescRet): ercType

where

pRwa

is the memory address of the Resource Work Area for the file that contains the resource.

typeCode

is the resource type code (word).

rsrcId

is the resource ID (word).

pbDesc

is the memory address of the resource descriptor format to which the returned information is written.

cbDescMax

is the maximum number of bytes to be written.

pcbDescRet

is the memory address to which the actual number of bytes written is returned.

Request Block

RsrcGetDesc is an object module procedure.

RsrcGetSetTypeInfo (pRwa, typeCode, cbPerDesc, pbDescBuf, cbDescBuf, pcDescRet): ercType

Description

RsrcGetSetTypeInfo writes in the buffer provided, all the descriptors of the specified type in the resource set identified by *pRwa* sorted numerically by resource ID. The caller must specify the size of the buffer to contain the descriptors and how many bytes of each descriptor are to be written. To calculate the buffer size, the caller can use the RsrcGetCountSetType operation. (See RsrcGetCountSetType.) If the buffer is too small to hold all the descriptors, the data is truncated.

RsrcGetSetTypeInfo differs from RsrcGetDesc in that the latter operation returns a desriptor for a single resource only.

If the file associated with the resource work area (RWA) does not contain any resources of the specified type, status code 52105 ("No such resource") is returned. If no resources exist in the specified file, status code 52012 ("No resources") is returned.

For details on the operations for managing resources in disk files, see "Accessing Resources in Disk Files" in the *CTOS Operating System Concepts Manual* section entitled "Utility Operations."

Procedural Interface

RsrcGetSetTypeInfo (pRwa, typeCode, cbPerDesc, pbDescBuf, cbDescBuf, pcDescRet): ercType

where

pRwa

is the memory address of the RWA.

typeCode

is the resource type.

cbPerDesc

is the number of bytes to be written for each descriptor.

pbDescBuf
cbDescBuf

describe the buffer to which the resource descriptor(s) are written. For details on the resource descriptor format, see "Resource Descriptor" in the *CTOS Procedural Interface Reference Manual* section entitled "System Structures."

pcDescRet

is the memory address of a word to which the count of resources of the specified type is written.

Request Block

RsrcGetSetTypeInfo is an object module procedure.

RsrcInitSetAccess (qFh, wMaskType, pRwaBase, pRwaBuf, cbRwaBuf, fModify, pFileType): ercType

Description

RsrcInitSetAccess initializes a buffer provided by the user as a resource work area (RWA) for the run file or resource binary specified by *qFh.* If the file is initially a zero-length file and the type specified by *qFh* is neither a run file nor a binary resource file, status code 52103 ("Unrecognizable file type") is returned. In subsequent resource management operations, the caller uses the memory address of the RWA as a handle to the resource set in the file.

The RWA is an in-memory work area used to maintain the status of the resource set in the file with which it is associated. RsrcInitSetAccess must be called to establish an RWA before an application can add or delete resources from the specified file or copy any of the file resources to another file.

If *qFh* is for a zero-length file, the user must specify the type of file to be created by setting bits in *wMaskType.* If no file type bits are set, status code 52103 ("Unknown file type") is returned. When an attempt is made to initialize a zero-length file with *fModify* set to FALSE, status code 52106 ("No such resource") is returned.

If the caller specifies *pRwaBase* as the handle of a base resource set and the buffer provided at *pRwaBuf* is not large enough to hold the resource descriptor table (RDT) it contains, status code 52109 ("RDT too large") is returned. In such a case, the required RDT size (in sectors) is returned at *pFileType*.

To be able to modify the resource set in the file, *fModify* must be set to TRUE, and the file must have been previously opened in modify mode. Otherwise, status code 52111 ("File unmodifiable") is returned.

For details on the operations for managing resources in disk files, see "Accessing Resources in Disk Files" in the *CTOS Operating System Concepts Manual* section entitled "Utility Operations."

Procedural Interface

RsrcInitSetAccess (qFh, wMaskType, pRwaBase, pRwaBuf, cbRwaBuf, fModify, pFileType): ercType

where

qFh

is the file handle (quad) of the file for which an RWA is to be established. The file must have previously been opened by either the OpenFile or the CdOpen operation. (See OpenFile and CdOpen.)

wMaskType

is a mask value (word) identifying the type of file opened. The values and their meanings are

Value	**Meaning**
4000h	File management file (opened by OpenFile)
8000h	CD-ROM service file (opened by CdOpen)

If a zero length file is being initialized, the caller must specify the file type to be made. See *pFileType* below.

pRwaBase

is the (optional) memory address of an existing RWA to be used as the base from which a new RWA will be constructed for the file specified by *qFh*. (*RwaBase* was returned in a previous call to RsrcInitSetAccess.) If this parameter is not used, it should be 0.

If a zero-length file is being initialized, the caller must specify the file type to be made. See *pFileType*.

pbRwaBuf
cbRwaBuf

describe the user-allocated buffer in which the RWA for the file specified by *qFh* is created. *cbRwa* must be an integral of 512 bytes and must be large enough to hold the resource descriptor table plus some additional data. The minimum size (in bytes) is 128 + (24*number of resources).

fModify

is a flag (byte). TRUE means resources can be added to or deleted from the file specified by *qFh*.

pFileType

is the memory address of a word identifying the disk file type as discovered by the operations looking at the file. The values and meanings of *FileType* are

Value	**Meaning**
1	Unknown
2	Binary resource file
4	Run file

Request Block

RsrcInitSetAccess is an object module procedure.

This page intentionally left blank.

RsrcSessionEnd: ercType

Description

RsrcSessionEnd closes an open temporary file created in a previous call to RsrcSessionInit. If any modified resource sets are still outstanding, RsrcSessionEnd calls RsrcEndSetAccess on them, causing the data referenced in the RWA to be written to the target files. An application should always call RsrcSessionEnd when it is finished using the resource management operations.

For details on the operations for managing resources in disk files, see "Accessing Resources in Disk Files" in the *CTOS Operating System Concepts Manual* section entitled "Utility Operations."

Procedural Interface

RsrcSessionEnd: ercType

Request Block

RsrcSessionEnd is an object module procedure.

This page intentionally left blank.

RsrcSessionInit (pbBuffer, cbBuffer, pbPswWrd, cbPswWrd): ercType

Description

RsrcSessionInit creates a temporary file to be used by subsequent resource file operations. The caller must allocate a buffer to be used by the resource operations for file I/O. Status code 52100 ("Bad buffer pointer") is returned if the memory address of the I/O buffer is invalid.

To use RsrcSessionInit, the caller must have a designated scratch volume upon which the resource operations can open a temporary file. If the scratch volume is password protected, the password must be provided. If a temporary file cannot be created, status code 52115 ("Cannot open scratch file") is returned.

For details on the operations for managing resources in disk files, see "Accessing Resources in Disk Files" in the *CTOS Operating System Concepts Manual* section entitled "Utility Operations."

Procedural Interface

RsrcSessionInit (pbBuffer, cbBuffer, pbPswWrd, cbPswWrd): ercType

where

pbBuffer
cbBuffer

describe the caller-allocated I/O buffer. *cbBuffer* must be greater than 4096 and a multiple of 512 bytes.

pbPswWrd
cbPswWrd

describe the scratch volume password that authorizes access to a specified file.

Request Block

RsrcSessionInit is an object module procedure.

43574656-100